National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

The University of Alberta

A HEURISTIC APPROACH TO QUERY OPTIMIZATION

by

© David J. Meechan

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-42824-4

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: David J. Meechan

TITLE OF THESIS: A Heuristic Approach to Query Optimization

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1988

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) David J. Meechan

Permanent Address:
4, 10654-83 Avenue
Edmonton, Alberta
Canada T6E 2E3

Dated 26 April 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Heuristic Approach to Query Optimization** submitted by **David J. Meechan** in partial fulfillment of the requirements for the degree of **Master of Science.**

Supervisor

Date 26 April 1988

ABSTRACT

Traditionally two approaches have been used by relational database management systems to determine the methods by which a query will be serviced. Either the cost of all applicable strategies are calculated and the minimum selected or strategies are given priorities and the applicable one with the highest priority is selected.

The approach discussed in this thesis, related to neither of the above, is heuristic. There are conditions under which a particular strategy offers the best performance. If this were not true then the strategy should not be implemented in the system to begin with. If those conditions can be identified then the selection of the optimal strategy is straightforward.

Acknowledgements

I would like to thank my supervisor, Tamer Ozsu, for his guidance and support throughout this research.

I would also like to thank Marjory Ross, for the support, sympathy, and tolerance expressed during my research.

Special thanks must go out to the other users of the systems who had to tolerate the running of the simulation program. In addition, I would like to thank the staff and patrons of the Power Plant, whose undying dedication to graduate student research, has created an ideal environment for advanced study.

Table of Contents

Chapter	Page
Chapter 1: INTRODUCTION	1
1.1. Database Overview	2
1.1.1. Logical View	2
1.1.2. Physical View	3
1.2. Query Processing Overview	4
1.3. Previous Related Work	7
1.4. Thesis Scope and Overview	9
Chapter 2: PERFORMANCE EVALUATION METHODOLOGY AND MODELS	12
2.1. Database Management System	12
2.1.1. Relational Data Subsystem	12
2.1.2. Relational Storage Subsystem	13
2.1.3. Scans	15
2.1.4. Joins	16
2.2. Simulation Models	17
2.2.1. Database Management System Model	17
2.2.1.1. Data Subsystem Component	17
2.2.1.2. Storage Subsystem Component	20
2.2.1.3. Buffer Manager Component	21
2.2.1.4. Sort Routine Component	21
2.2.2. Computer System Model	22
2.2.3. Database Model	23
2.2.3.1. Single Relations	23
2.2.3.2. Clustered Relations	23
2.2.3.3. Join Relations	24
2.2.4. Output Parameters	25
2.2.5. Parameter Values	25
2.3. Simulating Environment	27
Chapter 3: SIMPLE QUERY EXECUTION	28
3.1. Evaluation of Type 1 Queries	30
3.1.1. Effect of Buffer Size and Selectivity	30
3.1.2. Effect of Cluster Degradation	34
3.1.3. Summary	37
3.2. Evaluation of Type 2 Queries	38
3.2.1. Effect of Buffer Size and Selectivity	38
3.2.2. Effect of Cluster Degradation	42

3.2.3. Summary 46
3.3. Evaluation of Type 3 Queries 48
3.3.1. Effect of Buffer Size and Selectivity 48
3.3.2. Effect of Cluster Degradation 52
3.3.3. Summary 54
Chapter 4: JOIN EXECUTION 55
4.1. Joins Between Two Large Relations 58
4.1.1. One-to-One 58
4.1.2. One-to-Many 62
4.1.3. Many-to-Many 66
4.2. Joining a Large and a Small Relation 70
4.2.1. One-to-One 71
4.2.2. One-to-Many 77
4.2.3. Many-to-Many 82
4.3. Summary 85
Chapter 5: CONCLUSIONS 90
References 95
Appendix A 1: Simple Query Results 98
1.1. Type 1 Queries 98
1.2. Type 2 Queries 100
1.3. Type 3 Queries 104
Appendix A 2: Join Results
..... 109
2.1. Two Large Relations 109
2.1.1. 1-to-1 109
2.1.2. 1 to n 111
2.1.3. n to m 113
2.2. One Large and One Small Relation 115
2.2.1. 1 to 1 115
2.2.2. n to 1 117
2.2.3. n to m 119

List of Tables

Table	Page
2.1 Parameter Values	26
4.1 Join Methods	55
A1.1 Response Time vs Selectivity	98
A1.2 Confidence Interval for Response Time vs Selectivity	99
A1.3 Page Requests vs Selectivity	99
A1.4 CPU vs Selectivity	100
A1.5 Response Time vs Selectivity (1000)	100
A1.6 Response Time vs Selectivity (5000)	101
A1.7 Confidence Interval for Response Time vs Selectivity (1000)	101
A1.8 Confidence Interval for Response Time vs Selectivity (5000)	102
A1.9 Page Requests vs Selectivity (1000)	102
A1.10 Page Requests vs Selectivity (5000)	103
A1.11 CPU vs Selectivity (1000)	103
A1.12 CPU vs Selectivity (5000)	104
A1.13 Response Time vs Selectivity (1000)	104
A1.14 Response Time vs Selectivity (5000)	105
A1.15 Confidence Interval for Response Time vs Selectivity (1000)	105
A1.16 Confidence Interval for Response Time vs Selectivity (5000)	106
A1.17 Page Requests vs Selectivity (1000)	106
A1.18 Page Requests vs Selectivity (5000)	107
A1.19 CPU vs Selectivity (1000)	

107
A1.20 CPU vs Selectivity (5000)
108
A2.1 Response Time vs Join Selectivity
109
A2.2 Confidence Interval for Response Time vs Join Selectivity
110
A2.3 Page Requests vs Join Selectivity
110
A2.4 CPU vs Join Selectivity
111
A2.5 Response Time vs Join Selectivity
111
A2.6 Confidence Interval for Response Time vs Join Selectivity
112
A2.7 Page Requests vs Join Selectivity
112
A2.8 CPU vs Join Selectivity
113
A2.9 Response Time vs Join Selectivity
113
A2.10 Confidence Interval for Response Time vs Join Selectivity
114
A2.11 Page Requests vs Join Selectivity
114
A2.12 CPU vs Join Selectivity
115
A2.13 Response Time vs Join Selectivity
115
A2.14 Confidence Interval for Response Time vs Join Selectivity
116
A2.15 Page Requests vs Join Selectivity
116
A2.16 CPU vs Join Selectivity
117
A2.17 Response Time vs Join Selectivity
117
A2.18 Confidence Interval for Response Time vs Join Selectivity
118
A2.19 Page Requests vs Join Selectivity
118
A2.20 CPU vs Join Selectivity
119
A2.21 Response Time vs Join Selectivity

A2.22 Confidence Interval for Response Time vs Join Selectivity 119
A2.23 Page Requests vs Join Selectivity 120
A2.24 CPU vs Join Selectivity 120
..... 121

List of Figures

Figure	Page
2.1 System R	12
2.2 DBMS Model	17
2.3 System View	22
3.1 Query Type 1, Effect of Selectivity on Response Time	31
3.2 Query Type 1, Effect of Selectivity on Paging	31
3.3 Query Type 1, Effect of Selectivity on CPU Time	31
3.4 Query Type 1, Effect of Cluster Degradation on Response Time	34
3.5 Query Type 1, Effect of Cluster Degradation on Paging	34
3.6 Query Type 1, Limited Buffer Space, Effect of Cluster Degradation	36
3.7 Query Type 2, Effect of Selectivity on Response Time	39
3.8 Query Type 2, Effect of Selectivity on Paging	39
3.9 Query Type 2, Effect of Selectivity on CPU Time	39
3.10 Query Type 2, Limited Buffer Space, Effect of Selectivity	41
3.11 Query Type 2, Effect of Cluster Degradation on Response Time	42
3.12 Query Type 2, Effect of Cluster Degradation on Paging	42
3.13 Sort-Merge Buffering	43
3.14 Query Type 2, Limited Buffer Space, Effect of Cluster Degradation	44
3.15 Query Type 3, Effect of Selectivity on Response Time	48
3.16 Query Type 3, Effect of Selectivity on Paging	48
3.17 Query Type 3, Effect of Selectivity on CPU Time	

48
3.18 Query Type 3, Limited Buffer Space, Effect of Selectivity
51
3.19 Query Type 3, Effect of A_i Cluster Degradation on Response Time
52
3.20 Query Type 3, Effect of A_i Cluster Degradation on Paging
52
3.21 Query Type 3, Effect of A_j Cluster Degradation on Response Time
52
3.22 Query Type 3, Effect of A_j Cluster Degradation on Paging
52
4.1 Effect of Join Selectivity on Response Time
58
4.2 Effect of Join Selectivity on Paging
58
4.3 Effect of Join Selectivity on CPU Time
58
4.4 Effect of Join Selectivity on Response Time
60
4.5 Effect of Join Selectivity on Response Time
61
4.6 Effect of Join Selectivity on Paging
61
4.7 Effect of Join Selectivity on Response Time
62
4.8 Effect of Join Selectivity on Paging
62
4.9 Effect of Join Selectivity on CPU Time
62
4.10 Effect of Join Selectivity on Response Time
62
4.11 Effect of Join Selectivity on Response Time
65
4.12 Effect of Join Selectivity on Paging
65
4.13 Effect of Join Selectivity on Response Time
66
4.14 Effect of Join Selectivity on Paging
66
4.15 Effect of Join Selectivity on CPU Time
66
4.16 Effect of Join Selectivity on Response Time
66
4.17 Effect of Join Selectivity on Response Time

69
4.18 Effect of Join Selectivity on Paging 69
4.19 Effect of Join Selectivity on Response Time 71
4.20 Effect of Join Selectivity on Paging 71
4.21 Effect of Join Selectivity on CPU Time 71
4.22 Effect of Join Selectivity on Response Time 73
4.23 Effect of Join Selectivity on Paging 73
4.24 Effect of Join Selectivity on Response Time 75
4.25 Effect of Join Selectivity on Paging 75
4.26 Effect of Join Selectivity on Response Time 77
4.27 Effect of Join Selectivity on Paging 77
4.28 Effect of Join Selectivity on CPU Time 77
4.29 Effect of Join Selectivity on Response Time 79
4.30 Effect of Join Selectivity on Response Time 79
4.31 Effect of Join Selectivity on Response Time 81
4.32 Effect of Join Selectivity on Paging 81
4.33 Effect of Join Selectivity on Response Time 82
4.34 Effect of Join Selectivity on Paging 82
4.35 Effect of Join Selectivity on CPU Time 82
4.36 Effect of Join Selectivity on Response Time 84
4.37 Effect of Join Selectivity on Response Time 85
4.38 Effect of Join Selectivity on Paging 85

Chapter 1

INTRODUCTION

Relational database technology is being used by a growing community of end-users. Compared to more classical approaches, relational database management systems (DBMS) are known to significantly reduce the application development and maintenance times of information systems. This is primarily due to the high-level languages provided by such DBMS's. These languages are (more or less) non-procedural and use set-at-a-time operators. When querying a relational database, the user specifies the result wanted ("what") without providing the access paths to the data ("how"). The user does not need to know how the data is stored or the existence or availability of indices or any other (physical) organizational information; it is the responsibility of the database management system (DBMS) to resolve the "how". On the other hand, the non-procedural nature of these languages is one of the fundamental factors contributing to the difficulty of achieving high performance in execution. Performance typically is measured in one of two ways: *response time* and *resource utilization* (or *total time*). The DBMS has to select a method of executing the query; in fact, there may be several ways to service a query, and the component responsible for selecting the most appropriate strategy is the *query optimizer*. The conflict between high performance in execution and higher productivity with respect to system development and maintenance has resulted in considerable interest in the design of efficient query processing algorithms.

It is the goal of this thesis to investigate the practicality of a heuristic query optimizer. Typically, (often elaborate) cost functions are utilized in determining an optimum execution† approach. Such cost functions will include a variety of CPU and I/O costs often ignoring something as critical to I/O-intensive query processing as buffer space. Similarly, the algorithms used by the optimizer to determine an appropriate approach may be quite complex, causing a significant overhead in calculating these cost functions. In addition, there may be conditions under which the complexity and detail of such cost functions are not

† Even through the terms "optimum" and "optimization" are used frequently in relation to query processing, optimality in this context needs to be interpreted somewhat loosely. There is a significant amount of parameter estimation that is incorporated into the cost function parameters. Thus, the decisions are optimal to the extent that these estimates are accurate.

justified. Herein lies the thrust of our research and we pose the following questions: "Given a configuration and an operating environment, are there conditions which will indicate which of the available strategies is the better approach and, if so, what are they?" There are conditions under which one particular strategy performs better than the others. If this were not true, then the strategy could not have been implemented in the system to begin with. Therefore it should be possible, for a given environment, to construct an accurate query optimizer which bases its decision-making on those conditions rather than on the evaluation of elaborate cost functions.

1.1. Database Overview

Although it is assumed that the reader is familiar with database concepts to the extent discussed by Date [Dat77] and Ullman [Ull82], a brief overview will help define those areas of concern and establish a common terminology that we will use in the remainder of the thesis.

1.1.1. Logical View

At the logical level, a database provides strict access to the descriptions of instances of entities that it contains. This is done subject to the requirements of its community of users. An *entity* is a logical grouping of "things". Such things can be tangible or nontangible objects, such as a student, transcript or an event.

When we speak of an *instance* of an entity, we are referring to a member of such a grouping. Since such instances cannot actually be stored, a description of an instance is stored instead. An entity will be represented by a set of attributes, or selected characteristics, corresponding to real world properties. The description of an instance of an entity is a set of values associated with these attributes. For example, an employee may be represented by his name, his employee number, the department where he works, etc. An instance of an employee might be given by the description <Orville Buggywhip, 831430, Sales, ...>.

The logical structure of a database attempts to represent the interdependencies that characterize a complete picture of how this information is viewed by its community of users. Three data models are commonly used to represent this logical structure: hierarchical, network, and relational. Each data model

provides rules for grouping attributes. For example, the relational data model makes use of normalized relations. A relation is expressed in a normal form to remove certain undesirable properties such as data redundancy or attributes expressed as lists or sets of values.

1.1.2. Physical View

At the physical level, attributes are typically grouped to define database records. A *record-instance* at the physical level contains the description of an entity-instance at the logical level. The set of all record-instances for a particular record is called a *file*. Database records could be defined and their files stored according to groups specified by a data model; however, re-grouping at the physical level can be done to improve performance [Mar83].

Files reside permanently in secondary memory in groups called *data sets*. More accurately, a data set is a physical allocation of secondary memory in which some number of files are maintained. We use the term "physical allocation" to stress the fact that a data set is not necessarily a continuous portion of secondary storage. Each data set has a set of *access paths* associated with it. Access paths are algorithms and structures that are used to store and access the record-instances in the data set. For a good discussion of a variety of internal structures see [Bat85]. The types of access paths available is a design issue [SeC77, Yao77]. A data set with its access paths is called a *file organization*. A *physical database* is a set of interconnected file organizations. When the descriptions of related entities are stored in different data sets, interconnections must be maintained. The interconnections are maintained by pointers, where a pointer is any data item that can be used to locate a record-instance. Two types of pointers are most common: *symbolic pointers* and *direct pointers*. A symbolic pointer relies on an access path of the "pointed to" file organization to retrieve records. Direct pointers, on the other hand, reference record-instances by location.

Although access paths are concerned with records, the actual unit of transfer between secondary and primary memory is called a *block* (or sometimes a *page*). A block will typically contain more than one

record, the exact number being a design issue.

1.2. Query Processing Overview

The non-procedural nature of high-level relational languages dictates that the DBMS be responsible for determining how to access the data. The formulation of an approach to perform this task is done by a module referred to as the *query processor*.

For a given query, there often exists many strategies to perform it. The main objective of the query processor is to find the optimal approach with respect to the order in which operations are done as well as the utilization of access paths. The term "access path" is used in the database field to refer to the data structures and the algorithms that are used to access the data. A typical access path, for example is an index on one or more attributes.

The following example illustrates the importance of query optimization. Consider the well-known supplier-parts database consisting of three relations defined as follows:

P (PNO, PNAME)

S (SNO, SNAME, SADDRESS)

SP(SNO, PNO, QTY)

P is a parts relation consisting of attributes for a part number (PNO) and the part name (PNAME). S is a supplier relation consisting of attributes for a supplier number (SNO), the suppliers name (SNAME), and the address of the supplier (SADDRESS). The SP relation shows which part is supplied by which supplier and in what quantities (QTY). Assume that the cardinalities of relations P, S, and SP are 1000, 50, and 1000, respectively. This assumes that each part is supplied by a single supplier.

The following query is issued; "Names and addresses of suppliers who supply parts in quantities less than 10". Expressed in SQL the query is:

```
SELECT SNAME,SADDRESS
FROM P,SP
WHERE SP.SNO = S.SNO AND
      QTY < 10
```

Let us assume that there are 100 parts whose quantity is less than 10. Selections are performed by applying the predicate against each tuple, retrieving those tuples where the predicate is satisfied. In addition, joins are performed by a process of tuple substitution, where by the join attribute value from one relation is applied as a predicate against all tuples of the other relation.

There are two operations involved in this query: the selection of those tuples in SP which satisfy the predicate QTY < 10, and the join of the S and SP relations. Therefore, there are two ways to service this query. One way is to perform the join of the two relations and then perform the selection against the resulting relation. Each tuple of one relation must be compared with every tuple of the other relation requiring 1,000,000 comparisons. Performing the selection against this temporary relation requires a sequential access with 1,000,000 additional comparisons for a total of 2,000,000.

The second way to service this query is to perform the selection against the SP relation requiring 1,000 comparisons and resulting in a temporary relation of 100 tuples. Next the join is performed between this temporary relation (of cardinality 100) and the S relation requiring 5,000 comparisons for a total of 6,000 comparisons.

In addition to the two approaches offered above, there may be several ways to perform the join or several ways to retrieve tuples from a relation. The above example alone shows the importance of query optimization.

A query processor accepts as input a high level user query and generates low level data manipulation commands. In general query processing consists of two phases: compilation and execution. Compilation includes such tasks as normalization, semantic analysis, simplification and restructuring. Execution includes the tasks of the retrieval of data from storage and the performing of data manipulation operations.

During the normalization phase of compilation, the query is first lexically and syntactically analyzed and then transformed into a normalized form. Lexical and syntactic analysis is similar to that performed by compilers on general programming languages. Normal forms are either *conjunctive normal form* or *disjunctive normal form.*

The objective of semantic analysis is to refute incorrect queries. An incorrect query is one whose qualification can't be satisfied by any tuple. For example, the predicate (QTY > 10 AND QTY < 5) is contradictory and a query that includes it would be considered incorrect.

The simplification phase of compilation attempts to transform a query to a logically or semantically equivalent query that may be easier to execute. Logically equivalent queries are derived through the use the well known idempotency rules of boolean algebra. Semantically equivalent queries are derived through the use of integrity constraints. For example, if only the managers make more than 30,000 dollars then retrieving all managers is semantically equivalent to retrieving all employess whose wages are greater than 30,000 dollars.

The restructuring phase of compilation consists of constructing a query tree and modifying it. Operations such as selection and projection are pushed down the tree to ensure that they are performed as soon as possible.

The generation of low level data manipulation commands is associated with the process of optimization and can occur in either phase. If optimization occurs at compile time, (known as *static optimization*) the low level commands are expected to be stored and executed many times. Therefore the cost associated with optimization is expected to be amortized over many runs. In light of this, it is common to find an optimizer making use of exhaustive searches, complex cost functions, and a large amount of catalog information. System R [ABC76] and R* [MaL86b] are examples of this approach.

System R and R*, for local queries, makes use of a cost function of the form $Wcpu*(\#_instrs)+Wi/o*(\#_i/o)$ where *Wcpu* is the time to perform a CPU instruction and *Wi/o* is the time to fetch a page from disk. For a given strategy, the number of CPU instructions and the number of page fetches are estimated and assigned to *#_instrs* and *#_i/o* respectively. The approach with the minimum cost is selected as the execution strategy.

If optimization occurs at execution time, (known as *dynamic optimization*) the low level commands are expected to be executed immediately and only once. The minimization of the cost of optimization now

becomes critical. In such circumstances minimal catalog information is used and exact information on the intermediate relation sizes is used instead, as they are generated. The manner in which data is retrieved is typically determined by a priority ordering. INGRES [SWK76] is an example of such a system.

1.3. Previous Related Work

The use of heuristics in query optimization has, in the past, played a rather minor role. Many query optimizers make use of some form of heuristics to restrict the consideration of alternate strategies. King [Kin84] has introduced an approach called *Semantic Query Optimization* (SQO), whose purpose is to generate a semantically equivalent query that can be executed more efficiently than the original. Semantically equivalent queries are generated through transformations using the integrity constraints of the database. Making use of heuristics to guide this process restricts the generation of queries to those that may be more efficient to process. For example, a clustered index should always be exploited. Once these new queries are generated, each is passed to a conventional query optimizer. Estimated execution costs are retrieved and the query offering the least cost is the one selected.

Most of the current work in query optimization is directed towards the validation of cost functions [ASK80, MaL86a, MaL86b]. Some efforts have been made to improve the accuracy of these cost functions. A key component of such optimizers is the ability to accurately predict the number of tuples that will be retrieved. A value, called a selectivity factor, is used in the estimation. A selectivity factor represents the percentage of tuples retrieved. One of the first efforts at achieving this was done by Selinger and others [SAC79, SeA80] for System R. Assuming uniformity and independence of attribute values, they calculated the expected number of tuples retrieved using the size of the relation, cardinality of indices, and a set of functions based on operator types. Piatetsky-Shapiro and Connel [PiC84] have investigated this area further and have provided more accurate estimates using attribute value distribution. More recently, Choa and Egyhazy [ChE86] have made use of such statistics in estimating temporary file sizes in distributed databases.

Buffer pool size and management is critical factor affecting the performance of data retrieval. Some early investigations into the effect of main memory replacement algorithms on database performance were performed by Lang, Wood, and Fernandez [FLW78, LWF77]. Stonebraker [Sto81] has discussed the conflicting goals between buffering in an operating system environment and in a database environment. Locality of reference, characteristic of programs, leads to good buffer management with a least recently used (LRU) policy, but is not characteristic of database accesses. The conclusion is that there is a need for greater communication between a DBMS and an OS providing buffer management.

Both INGRES [SWK76] and System R [ABC76] have opted to put buffer pools in user space to enhance performance by reducing system access overhead. More recently, work has been done on buffer management algorithms tailored for database application; for example, Sacco and Schkolnick's Hot-set model [SaS86], Chou and DeWitt's DBMIN algorithm [ChD86], and the work done by Effelsberg and others [EfL84, EfH84].

Clustered indices [BlE77, SAC79] can be used to reduce the number of I/O operations, since the tuples of a relation are inserted into physical pages according to an index ordering. A clustered index has the property that not only each index page, but also each data page containing a tuple from that relation will be accessed only once during a scan on that index. Therefore, several tuples can be retrieved per page access. Bergamaschi and Scalas [BeS86] have extended this approach over multiple indices.

Unfortunately, the cost of ensuring that the indices remain clustered after every update is quite high and the ordering is maintained by periodic reorganization. In between reorganizations, the ordering of the tuples degrades, resulting in the loss of the property that each page is accessed only once. These perturbations cause the system performance to approach that of the random tuple placement. The study of the system behavior as a result of cluster degradation is important because it directly affects the effectiveness of the access path selection heuristics. An optimizer whose cost function is designed to be sensitive to the clustering of indices will inevitably make poor decisions if the clustering degrades. On the other hand, by considering tuple placement as random and disregarding the potential savings of a pseudo-clustered index,

an optimizer may overlook an index as an efficient access path.

A variety of approaches to improve the performance of query execution with additional or unique hardware have been proposed. These include the use of multiple processors [BBD83], intelligent terminals [HaS79], database machines [DeW79], and back-end machines [HaF86]. These may well improve the performance of query execution, however the DBMS will still have to select an approach to be used and its performance will still be subject to this selection. Given a configuration and an environment, whether optimal or not, the query optimizer should still select a good approach.

In addition, approaches to performing joins have received much attention. These include hash joins [Sha86] and join indices [Val87] as well as join processing in DBMS with large main memory [Sha86] and main memory DBMS [LeC86].

1.4. Thesis Scope and Overview

Query optimizers generally make use of one of two approaches: priority schemes and cost functions. Priority schemes are quick and easy to use and are typically implemented in systems that interpret queries. Low overhead of optimization is the goal motivating their use. However, priority schemes offer little flexibility. The order is dictated by what is good, or what is thought to be good, in general. under certain conditions such an ordering may prove to provide a very poor strategy.

Cost functions, on the other hand, are used by systems where once a query is compiled it is expected to be executed many times. Overhead of optimization is less critical as it is expected that the optimized query will be run many times, amortizing the cost.

In either case, the effect of buffering is not considered by the query optimizers. Queries are inherently I/O intensive operations. Disregard for such a critical component can cause either approach to offer poor choices.

The first step is to develop a model of a set configuration and operating environment. By investigating the response time of each execution strategy, for a given query type, under various conditions we hope

to derive heuristic rules that, when satisfied, will determine the approach that should be used.

Referring again to Mackert and Lohman [MaL86b], the R* optimizer correctly chooses clustered indices whenever they exist. If using clustered indices is preferred whenever they exist, then one has to question the validity of calculating a variety of cost functions to arrive at that decision. It would seem to make more sense to have the optimizer select a clustered index if it exists and not bother with evaluating other processing strategies. As well, it is reported that, during a join, if an unindexed inner relation cannot fit entirely into the buffer, the optimizer correctly estimates *in all circumstances* that a nested loop join is hopelessly expensive. If this is true then why is the optimizer even considering nested loop joins in this case?

Are there more of such circumstances? The goal here is to determine and derive relatively simple heuristics that may be utilized by query optimizers and run-time routines, thereby, simplifying the selection process of the query optimizer while still maintaining flexibility in determining access for individual cases.

A number of retrieval and join techniques for a variety of query types will be considered. Although some of the results are well known, they have not previously been taken full advantage of. Our research deals only with centralized database systems. The issues in distributed systems are considerably more complicated [OzV88].

Because each system offers different query processing strategies and is based on different architectures the manner in which to service a query will vary from system to system. In light of this, our investigation is based on a specific system and specific architecture. In this environment we study a variety of query processing techniques subject to various query types through simulation. The simulation model has been validated by comparing the results against reported actual measurements. Using these performance results, we derive simple heuristics.

Chapter 2 contains the description of the data base system on which the simulation model has been based. Emphasis has been placed on those components and activities involved in the retrieval of data. The simulation model itself is presented in Chapter 2 as well. Chapter 3 looks at simple queries, those that

access a single relation, while Chapter 4 investigates joins. Finally, Chapter 5 presents some conclusions and suggests directions for future research.

Chapter 2

PERFORMANCE EVALUATION METHODOLOGY AND MODELS

In this chapter we present the model that has been developed to simulate query execution. The database system we have chosen as a base for our model is R* [Dan82, DSH82, MaL86a, MaL86b, Ng82], which is an extension of System R [ABC76]. Although R* is a distributed database we are only concerned, in this thesis, with its local processing capabilities. This particular DBMS was chosen because of the ample amount of detailed information which is available in the literature and because of the accuracy of the cost functions utilized by the optimizer. Furthermore, there exists sufficient performance data about these processing techniques to establish the validity of our model.

We begin in Section 2.1 with a brief review of the components that are directly related to the utilization of access paths at a single site. Section 2.2 presents the simulation model that has been developed to simulate this system as well as the assumptions and the metrics that are used.

2.1. Database Management System

R* at a site, depicted in Figure 2.1, is composed of two major components: the *Relational Data Subsystem* and the *Relational Storage Subsystem*. Users interact with the Relational Data Subsystem through a Relational Data Interface. The Relational Storage Subsystem provides a tuple-at-a-time interface to the Relational Data Subsystem by a Relational Storage Interface.

2.1.1. Relational Data Subsystem

The Relational Data Subsystem (RDS) is the interface for the outside world to access the stored data, performing the authorization checking and integrity enforcement functions. Catalogs of external names are maintained by the RDS since all lower-level components operate entirely on internally generated names. In addition the RDS provides alternative views of the data.

Users may call the RDS directly via the Relational Data Interface (RDI) which supports the high level language SEQUEL. The RDS contains an optimizer which, given an RDI command, derives a plan

for its execution. The optimizer will select an access path to the data from those that are provided.

2.1.2. Relational Storage Subsystem

The Relational Storage System (RSS) provides the access to the data for System R. Communication between the RDS and RSS is through the Relational Storage Interface (RSI). This interface provides tuple-at-a-time operations on the base (or actual) relations.

The main data object of the RSS is the n-ary relation, which consists of a time-varying number of tuples, each containing n fields. Associated with every tuple of a relation is a tuple identifier (or TID). Each TID is generated by the RSS and is available to the RDS as a concise and efficient means of addressing tuples. TIDs are also used within the RSS to refer to tuples from index structures, and to maintain pointer chains. However, they are not intended for end users above the RDS, since they may be re-used by the RSS after tuple deletions and since they are reassigned during database reorganization.

Each tuple is stored as a contiguous sequence of field values within a single page. Field lengths are also included for variable length fields. A prefix is stored with each tuple for use within the RSS. The prefix contains such information as the relation identifier, the pointer fields (TIDs), the number of stored data fields, and the number of pointer fields. Tuples are found only on pages which have been reserved as data pages. Other pages are reserved for the storage of index or internal catalog entries. A given data page may contain tuples from more than one relation, so that extra page accesses can be avoided when tuples from different relations are accessed together.

Each index is composed of one or more pages. The pages for a given index are organized into a balanced hierarchical structure in the style of B-trees [BaM72]. Each page is a node within the hierarchy and contains an ordered sequence of index entries. For nonleaf nodes, an entry consists of a <sort value, pointer> pair. The pointer addresses another page in the same structure, which may be either a leaf page or another nonleaf page. In either case the target page contains entries for the sort values less than or equal to the given one. For the leaf nodes, an entry is a combination of sort values along with an ascending list of

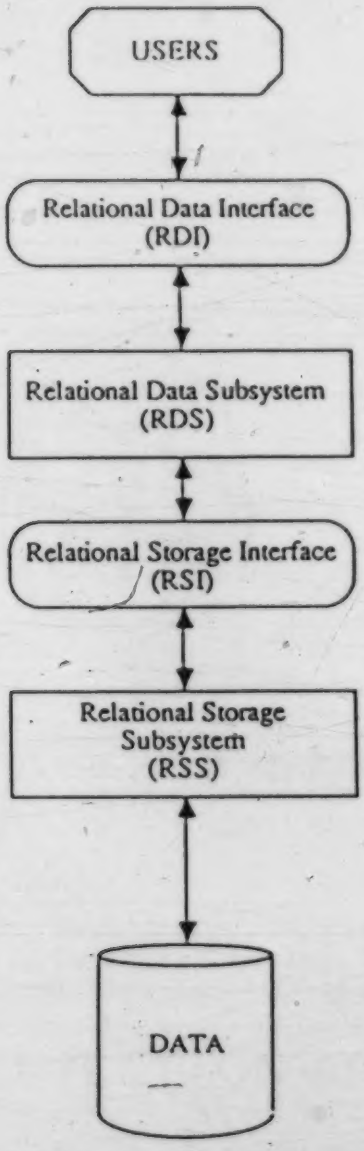

Figure 2.1 System R

TIDs for tuples having exactly those sort values. The leaf pages are chained in a doubly linked list, so that sequential access can be supported from leaf to leaf.

RSS maintains these two types of pages: data pages and index pages, by grouping them into logical units called segments. All pages are of equal size. Although a given segment may contain several relations, relations may not span segments.

A page request is handled by allocating space within a main memory buffer shared among all concurrent users. Pages are fixed in their buffer slots until they are explicitly freed by RSS components. Freeing a page makes it available for replacement, and when space is needed, the buffer manager replaces whichever freed page was least recently requested.

2.1.3. Scans

One can fetch a sequence of tuples along an access path through the use of an RSS cursor or scan. Each scan is created by the RSS for fetching tuples on a particular access path via the execution of the OPEN_SCAN operator. The tuples along the path may then be accessed by a sequence of NEXT operations on that scan.

For all of these access paths the RDS may attach a search argument to each NEXT operation. The search argument may be any disjunctive-normal-form expression where each atomic expression has the form <field#, operator, value>. The value is an explicit character string provided by the RDS, and the operator is one of =, ≠, <, >, ≤, or ≥.

A dbspace scan is a sequential scan of tuples. When a dbspace scan is executed on a relation, an internal scan is generated on all nonempty data pages within the segment containing the relation. Each such data page is touched once, and the prefix of each tuple within the page is checked to see if it belongs to the relation.

An index scan examines the leaf pages of an index and then access all records, in a given range, by TID. When an entire relation is retrieved, each page of the index is touched only once, however, a data

page may be accessed several times.

Tuples may also be retrieved in an ordering determined by the sorted order of a field or combination of fields. In this type of access, a relation is read directly into a sort routine and a temporary relation is generated with tuples sorted on those fields specified. This temporary relation is then accessed via a sequential scan. The columns to be sorted in each tuple are encoded before sorting and decoded after sorting. Each page is initially quicksorted and then an n-way merge is performed where $n \geq 8$.

2.1.4. Joins

Joins are performed using one of two methods: *nested loop* or *merged scan*. A single relation, called the inner relation, is joined with another, possible composite relation called the outer relation. Joins involving multiple relations are performed by using the result of one join as the outer relation of the subsequent join.

During a nested loop scan, for each tuple of the outer relation retrieved, a completely new scan is opened on the inner relation. The scan of the inner relation will either be a dbspace scan or an index scan. A new scan must be opened to re-set all internal pointers to the beginning of the file or index since there is no guaranteed ordering.

A merge scan join requires that both relations be accessible in join attribute order. Pointers are maintained at the previously referenced tuple. As such, for each tuple of the outer relation, a complete scan need not be performed. Suppose that two relations, R1 and R2, are sorted in ascending order with respect to the join attribute a, two cursors $c1$ and $c2$ identify a tuple in each relation. The cursors are positioned on the first tuple of each relation initially. If $c1.a = c2.a$ then $c1.a$ is concatenated with $c2.a$ and the result is returned and $c2.a$ is advanced to the next tuple. If $c1.a > c2.a$ then $c2$ is advanced, otherwise, $c1$ is advanced.

In addition, a "*back cursor*" is maintained on the inner relation, asy R2. This cursor is positioned on the first tuple of any sequence containing the same a value. If a new R1.a value is equal to the previous

value, *c2* is reset to the position of the back cursor. This process continues until one of the relations is exhausted.

2.2. Simulation Models

The simulation model can be decomposed into three components: the Database Management Model, the Computer System Model, and the Database Model. The Database Management Model represents the software running on a computer system, represented by the Computer System Model. The Database Management Model acts upon relations and index structures represented by the Database Model. The simulator operates on actual statistical data which is generated in advance.

2.2.1. Database Management System Model

Figure 2.2 depicts the components of the database management system model. This model includes the Data Subsystem component (DS), the Storage Subsystem component (SS), the buffer manager component, and a sort routine component. The DS and SS correspond to the Relational Data Subsystem and the Relational Storage Subsystem, respectively, which are described in more detail below.

2.2.1.1. Data Subsystem Component

When performing a scan on a single relation the Data Subsystem (DS) communicates with the Storage Subsystem (SS) using four commands; OPEN_SCAN, CLOSE_SCAN, NEXT, and FETCH. The OPEN_SCAN operator causes the SS to create a scan on a specified access path. Once a scan has been opened, tuples along the path may be accessed by the NEXT operator while the FETCH operator retrieves any combination of fields within a tuple.

The DS models the retrieval of tuples through a generic access path by the following sequence of commands that is expressed in pseudo-code.

```
OPEN_SCAN(relation, scan_type)
WHILE NOT end of scan DO
BEGIN
    NEXT(operator, value)    ! locate tuple matching the predicate
```

```
    FETCH                ! retrieve tuple
    apply residual predicate
END
CLOSE_SCAN
return result
```

Regardless of the access path being used, the same sequence of events occurs. Three types of scans are currently permitted; a *dbspace* (or *segment*) *scan*, an *index scan*, and a *sort scan*. The operator may be one of "=", ">", or "<" with "value" being a character string or numeric value as the case may be.

When performing joins, the DS communicates in a similar fashion. A scan is opened on the outer relation and for each tuple retrieved, a scan is opened on the inner relation. If a nested loop is being used, the scan of the inner relation is started from the beginning of the file. If a merged scan is being employeed, the scan of the inner relation begins at the previously accessed point.

DS models a join operation using the following sequence of commands embedded in pseudo-code.

```
OPEN_SCAN(outer_relation, scan_type)
WHILE NOT end of scan DO
BEGIN
    NEXT(operator, value)    ! locate tuple matching any predicate
    FETCH                    ! retrieve tuple
    OPEN_SCAN(inner_relation, scan_type)
    WHILE NOT end of scan DO
    BEGIN
        NEXT(operator, join_key)    ! locate matching tuple
        FETCH                ! retrieve tuple
        apply residual predicate
    END
    CLOSE_SCAN
END
CLOSE_SCAN
return result
```

The parameters used to characterize the relational data subsystem model consist of the time required to apply a residual predicate and the time required to return a result to the application program's address space. A residual predicate may be one to retrieve the average, minimum, or maximum of some field. The following assumptions are made about the DS:

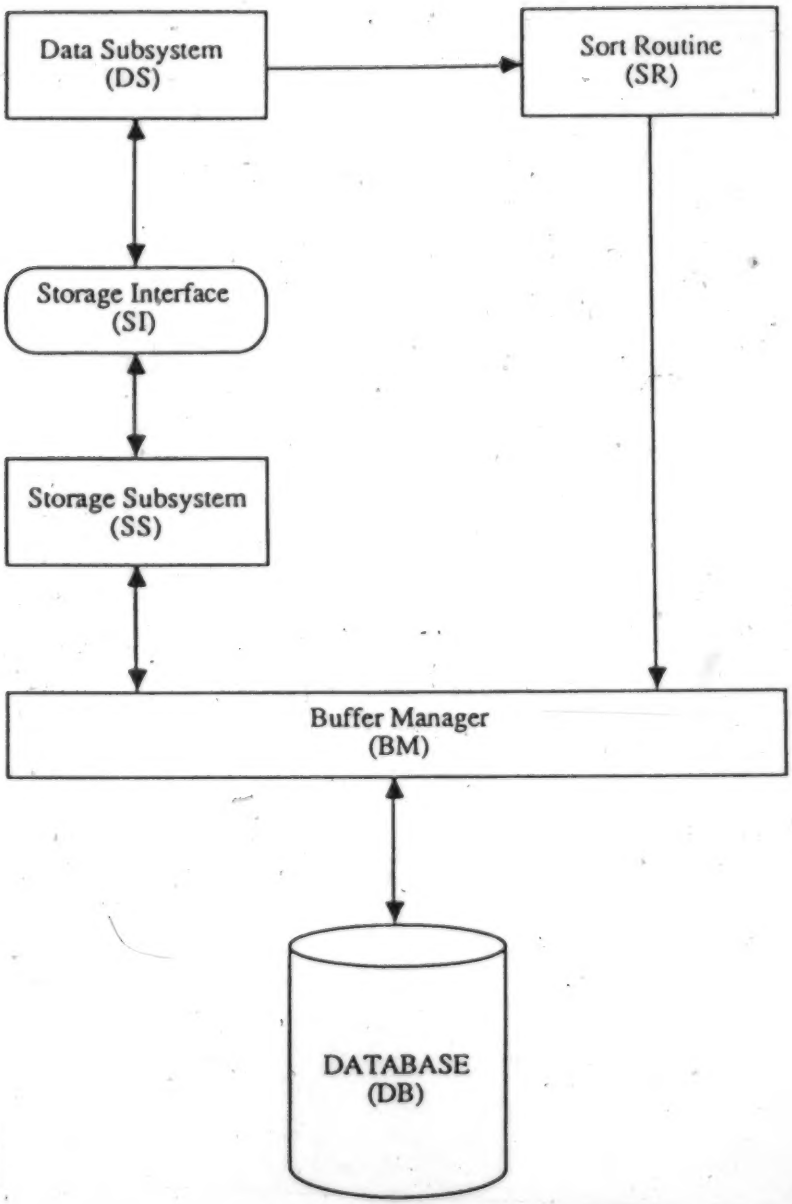

Figure 2.2 DBMS Model

a) The CPU time required to apply a residual predicate (*resp*) includes any overhead and is a function of the number of expressions in the predicate.

b) The CPU time required to return a result to the application program's address space (*return*) is a constant. However the number of times it is applied is a function of the result size and the degree of blocking.

2.2.1.2. Storage Subsystem Component

The Storage Subsystem (SS) acts upon the commands issued by the DS and the actions taken are dependent on the type of scan requested. The scans are the basic operations of data retrieval.

When a dbspace scan is opened, a tuple extraction is performed for each data page. The predicates are applied to each tuple on a page. When an index scan is opened, a tuple extraction is performed twice for each tuple accessed: once to access the index and a second to access the tuple. A sort scan is, in fact, two dbspace scans. The first reads the tuple directly into a sort routine which will produce a temporary relation in sorted order. The second scan is opened on this temporary relation. When a merged scan join is performed, the position within the inner relation must be maintained.

The parameters used to characterize the storage subsystem model consist of the following:

a) The CPU time required to access a page in the buffer and extract the desired tuple (*btuple*).

b) The CPU times required to open and close a scan are different for index (*openi* & *closei*) and dbspace scan (*opend* & *closed*).

c) The CPU time required to apply a single predicate (*singlep*).

d) The CPU time to invoke the SS for a dbspace scan and for an index scan (*invoked* & *invokei*). Note that a sort scan will involve one of these on the initial access, and a dbspace scan on the sorted relation.

e) The CPU time to invoke the SS for a dbspace scan and for an index scan (*minvoked* & *minvokei*) on the inner relation during a merged scan. These times include the overhead of maintaining pointers.

2.2.1.3. Buffer Manager Component

The buffer is shared among all concurrent processes. The page replacement algorithm is Global Least Recently Used (GLRU). The parameter that characterizes the buffer manager is the buffer size (*Bs*) given as the number of pages in the buffer pool.

If a referenced page is not in the buffer, a disk read request must be issued. During a write operation, the page is first written to the buffer and then a disk write request is issued, causing this page to be queued immediately. Since most DBMS's use demand paging, our simulation model implements the same strategy. The issue of prefetching is independent of this study although its effects may be of some importance. The current study does not investigate other buffering schemes and, as such, only the one described has been modeled.

2.2.1.4. Sort Routine Component

When an ordering is requested in a query or the relation will take part in a merge scan join, the result relation must be in sorted order. When sorting, each page is initially quicksorted [AHU74, MaL86b] and then an n-way merge is performed where $n \geq 8$ as described in the Section 2.1.3. The columns to be sorted in each tuple are encoded before sorting and decoded after sorting.

The parameters used to characterize the sort routine consist of the following:

a) The CPU time required to open and close (*opent* & *closet*) a temporary relation in which to store temporary results.

b) The CPU times required to encode and decode the fields to be sorted (*encode* & *decode*) is a linear function of the number of fields.

c) The CPU time required to quicksort a page (*quicksort*).

d) The CPU time to access a page in the buffer (*bpage*).

e) The CPU time required to compare the values of one tuple to be merged (*tcomp*).

2.2.2. Computer System Model

A site is viewed as containing two resources; a CPU and a disk. Associated with each resource is a queue which is managed according to First-Come-First-Serve (FCFS) with multiplexing discipline. Thus the processes are granted a resource on a first-come-first-serve basis by the respective servers which are, at the same time, multiplexed among different processes, enabling multiprogramming. Although no time slicing exists, multiplexing may take place either when a process is waiting for data from another process or as a result of a disk request. Figure 2.3 shows the system view. Although this view is valid for the query as a whole, it is, in fact, modeled as two processes: one for the RDS and the other for the RSS. Only the RSS process has access to the disk. The RDS process issues commands to the RDS and waits for a response.

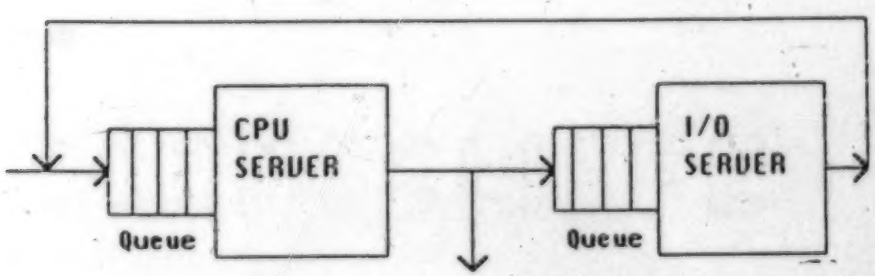

Figure 2.3 System View

The only other parameter used to characterize the operations of the computer system is the I/O time to read/write a page from/to disk (*IO*). Times required for various CPU operations vary with the type of action and are more closely related to the components (i.e. data subsystem and storage subsystem) that issue them. Therefore, they were discussed in the previous section.

2.2.3. Database Model

The database model is represented by a sequence of relations where each relation is composed of data and index structures. Tuples of a relation consist of one or more attributes which are integer values. These values, viewed as the only relevant fields, are the sole target of predicates and are the field upon which indices are built. Tuple length, in itself, is not a concern in this study since the cost of moving data within memory is considered negligible, which is consistent with other studies. However, tuple length is represented as the number of tuples per page, which is set at 50 tuples per page. Furthermore, each relation is assumed to be the sole occupant of its segment. Index tuples are composed of a <field value, page location> pair. The database is generated prior to the simulation run and queries operate on actual data.

2.2.3.1. Single Relations

The generation of single relations, the target of single relation queries, is relatively straightforward. Attribute values are unique and range from 1 to the relation cardinality. This is purely an implementation decision to facilitate the control of the number of tuples to be retrieved. Given a relation's cardinality, tuples are randomly assigned to pages. Index tuples are then constructed using the attribute value and its assigned page number. If more than one attribute is being generated, each one is handled independently in the same manner. As an example of this generation, consider a single relation containing 1000 tuples. The numbers 1 to 1000 are randomly assigned to 20 groups of 50. Each group of 50 constitutes a page. Now a selection requesting all tuples with attribute values less than 501 guarantees a selectivity of 50%.

2.2.3.2. Clustered Relations

A clustered relation, or more accurately, a clustered index, has the property that not only will each index page be touched at most once, but so too will each data page. This is achieved by maintaining a physical proximity of tuples corresponding to the index ordering.

Clustered relations are generated in similar fashion to those described previously. However, tuples are assigned to pages in numerical order, instead of a random placement. In other words, the tuples are

inserted into pages in the index order.

Of interest in this study is the degradation of clustered indices. By degradation we mean disruption of the physical proximity of tuples that is characteristic of clustered indices. To model this, we introduce a function SWAP. Each time SWAP is called it randomly selects two tuples and switches their locations thereby degrading the physical ordering of the relation while maintaining control over the selectivity. For example, given a completely clustered relation such as:

	PAGE 1					PAGE 2				
LOCATION	1	2	3	4	5	1	2	3	4	5
VALUE	1	2	3	4	5	6	7	8	9	10

If the two tuples selected are the third tuple of page 1 and the fourth tuple of page 2 then the resulting relation following one execution of SWAP is:

	PAGE 1					PAGE 2				
LOCATION	1	2	3	4	5	1	2	3	4	5
VALUE	1	2	9	4	5	6	7	8	3	10

Therefore, by increasing the number of SWAPs performed, a gradual degradation of the physical ordering can be achieved. It should be noted that only one index can be clustered at any given time.

2.2.3.3. Join Relations

One aspect of join operations under investigation is join selectivity. Join selectivity, may be defined as $card(RJNS) / card(R) * card(S)$ for two relations R and S where JN is the join operator. If the join attributes are randomly generated, little control over the join selectivity can be had. In order to manipulate this parameter we must control the cardinality of the resulting relation relative to the cardinality of the two original relations.

When generating join relations, we first identify two values: the number of occurrences of a unique join attribute value in both the inner relation (denoted n) and the outer relation (denoted m). In addition, we introduce the value α, which is the number of occurrences of an n,m match, so that the cardinality of the

result relation will be given as $\alpha * (n * m)$. Therefore, by controlling these the values, we control the cardinality of the result relation. As an example, consider two 1000 tuple relations. If n=2, m=5, and α=100, then the cardinality of the resulting relation will be $\alpha * (n * m) = 1000$ and the join selectivity will be 0.001.

2.2.4. Output Parameters

The output parameters that we consider in this study are the response time, CPU time, and the time spent on I/O. Although total time, as a summation of component usage time, could be an additional metric for this study, we have not explicitly measured it since it is in fact very close to response time. Two reasons for this assumption are:

1) Queries are executed in isolation and, although two processes are running, they are all highly dependent on one another. A process releases a resource to wait for a process that returns some data. There are no other processes that will acquire the resource.

2) Although a process need not wait for the completion of a write, most writes are closely followed by a read. Writes only occur during sorting at which time the event of a page being filled will, in general, be relatively close to the time of a page being emptied and a subsequent page being read. Some disk controllers do maintain two I/O queues giving priority to reads (which block processes), but this has not been modeled in the current study.

2.2.5. Parameter Values

The level of detail of the models discussed is based on the cost functions of System R. Although no values for these parameters have been reported or made available, the resulting response times for some access paths and joins have. Therefore, by first determining the sequence of operations, we have worked backwards from the response times to get approximations to these values.

These approximations may, however, be distorted; but should pose no significant problem since, for these access paths and joins, certain collections of parameters occur with the same frequency. Although the

individual times may be in error, the collective weight has proven to be accurate. For example, an index scan requires the use of the following [MaL86a];

$$(openi + closei) + SEL_i * CARD * (2 * btuple + singlep) + SEL_t * CARD * (invokei + resp).$$

The first term represents the cost of opening and closing the scan. In the second term SEL_i is the selectivity of the predicate applied to the index. 2**btuple* is the cost of accessing both an index page and a data page in the buffer and *singlep* is the cost of applying a SARGable predicate. Where a SARGable predicate is a boolean predicate evaluating to a true or false value such as *Name=Fred*. In the third term SEL_t is the selectivity factor of all SARGable predicates. *invokei* is the cost of invoking the RSS and *resp* is the cost of applying the residual predicates. The response times reported are the result of applying a single predicate which means $SEL_i = SEL_t$. By using just *SEL* the above formula can be reduced to,

$$(openi + closei) + SEL * CARD * (2 * btuple + singlep + invokei + resp)$$

We can clearly see that the individual values of *btuple* , *singlep* , *invokei* , and *resp* are less critical then the weight of the whole. Table 1 presents a review of the parameters as well as approximate values. Those parameters under investigation will be defined and discussed in those sections concerned with their effect.

PARAMETER	VALUE	PARAMETER	VALUE
IO	17ms	tcomp	0.025ms
opend	1.488ms	closed	1.488ms
btuple	0.132ms	singlep	0.02ms
invoked	0.42ms	invokei	0.36ms
minvoked	0.61ms	minvokei	2.7ms
openi	3.7ms	closei	3.7ms
return	16.5ms	resp	0.197ms
opent	40.5ms	closet	40.5ms
encode	2.5ms	decode	2.5ms
quicksort	5ms	bpage	0.150ms

Table 2.1 Parameter Values

2.3. Simulating Environment

The simulator is written in the high-level language SIMULA. Experiments where run under the UNIX 4.3BSD operating system on a VAX 11/780.

The results were obtained from simulation runs over 16 different databases generated from different seeds. To give an accuracy of the results, 95% confidence intervals are computed for the mean response times R as:

$$\left[R - 1.96\sqrt{s^2/n}\ ,R + 1.96\sqrt{s^2/n}\right]$$

where s^2 is the sample variance obtained from the simulation and n is the total number of samples. Each sample was generated independently.

Chapter 3

SIMPLE QUERY EXECUTION

In this chapter we concentrate only on simple queries, which access a single relation. Such queries are specified by using unary relational algebra operations such as selection and projection. The discussion of the results of the experiments investigating the evaluation of joins is left to Chapter 4.

In order to investigate the performance of all access paths available, two forms of simple query are considered. The first is a simple retrieval where no ordering of the result relation is requested. The second form requires that the result relation be in sorted order. The first form can be serviced with either an index scan or a dbspace scan. The second, because an ordering is requested, must either be serviced by an index scan or a sort scan. When a simple query includes a request for an ordering, the attribute targeted for ordering may not be the attribute targeted by the predicate. To facilitate this situation the second form of the simple query is represented by two types: the first where the ordering is requested on the predicate attribute and the second where the ordering is requested on an alternative attribute.

In the following, we assume that R is a relation defined over the set of attributes $A = \{A_1, A_2, ..., A_n\}$ (usually denoted $R[A_1, A_2, ..., A_n]$). The example relation that we will use is one which stores information about the employees in an organization. The relation can be defined as EMPLOYEE(ENO,ENAME,DEPT,SALARY), where ENO is the employee number, ENAME is the employee name, DEPT is the employee's department and SALARY represents his/her salary. The types of queries that are considered are as follows :

(1) **Type 1 - Simple Selection :** This is a query of the type $A_i\ \theta\ value$ where $\theta \in \{<, >, =, \leq, \geq, \neq\}$. It involves a selection over a relation where the selection predicate on R is defined over a single attribute.

An example would be to retrieve the names and salaries of employees in the engineering department. In SQL [Dat87] notation, it can be specified as

```
SELECT ENAME,SALARY
FROM EMPLOYEE
WHERE DEPT="Engineering"
```

(2) **Type 2 - Selection with Ordering of Predicate Attribute :** This type of query is the same as the first type, but an ordering of the resulting relation is requested on A_i which is the attribute involved in the selection predicate. For example, the following query requires the result to be sorted on the SALARY attribute over which the selection is made.

```
SELECT ENAME,SALARY
FROM EMPLOYEE
WHERE SALARY>20000
ORDER BY SALARY
```

(3) **Type 3 - Selection with Ordering of Alternate Attribute :** This type of query is similar to the second type. However, the ordering of the resulting relation is to be performed on some attribute A_j ($j \neq i$) which does not participate in the selection predicate. An example for this type of query is the following where the resulting relation is sorted on the SALARY attribute, but the selection is done over the DEPT attribute.

```
SELECT ENAME,SALARY
FROM EMPLOYEE
WHERE DEPT="Engineering"
ORDER BY SALARY
```

The results of our experiments on the effects of buffer size and selectivity for the three types of queries are presented in Sections 3.1, 3.2, and 3.3. Selectivity is the percentage of tuples retrieved from the relation. Retrieving 500 tuples from a 1000 tuple relation represents a selectivity of 50%. In each section the response times for the methods of servicing the query are analyzed and trends identified. Based on these observations, a number of heuristic rules are derived. In light of the fact that these are based on a specific configuration and implementation, potential improvements will be suggested where applicable. These modifications may, in fact, alter the results presented.

3.1. Evaluation of Type 1 Queries

The queries of this type are very simple and sufficient knowledge exists within the database community regarding the behavior of query processors in executing them. The fundamental reason for including Type 1 queries in our study is completeness and the fact that the results form the basis for the discussion of the behavior of the other query types. Since no ordering is required, the tuples may be retrieved by using either a dbspace scan or an index scan.

The results of the experiments with buffer size and selectivity are reported in Section 3.1.1 while the effects of clustered index degradation are presented in Section 3.1.2. Section 3.1.3 presents a summary of the conclusions drawn.

3.1.1. Effect of Buffer Size and Selectivity

When the selectivity is sufficiently low, an index scan will always offer superior performance. In this context, a "sufficiently low" selectivity refers to the number of retrieved tuples being less than the total number of pages. In other words, even in the worst case, the number of pages retrieved by an index scan is less than the number of pages retrieved by a dbspace scan which accesses each page only once. This holds true regardless of buffer size.

If the selectivity is such that all the data pages will be accessed at least once, a dbspace scan will always offer superior performance, again regardless of the buffer size. This is because, even if the buffer size is large enough to hold all the index and data pages, both an index scan and a dbspace scan will access the same number of data pages. However, additional I/O is required during an index scan to retrieve the index pages. In the event that buffer space is limited, random probing of the index scan will cause page faults and additional I/O.

The optimal access path, between these two extremes of low and high selectivity, is a function of the buffer size and the number of pages accessed. A query selecting n tuples from a relation of p_d data pages and p_i index pages does not necessarily access $\min(n, p_d)$ data pages. Some tuples may reside on the same

page. The actual number of data pages accessed (p_d') can be estimated by a function given by Yao [Yao77a] or an approximation to that function as described by Bernstein [BGW81]. If p_d' is less than the buffer size (b) and $p_d' + p_i < p_d$, an index scan is preferred. However, if p_d' is greater than b, then extra page faults due to random probing may occur during an index scan.

If $b \leq p_d' + 1$, the amount of paging required can be approximated by

$$b + (2n - b)(1 - b / (p_d' + 1)) \qquad \text{(Eq. 3.1)}$$

assuming that the first b requests are unique and therefore fill the buffer. The probability that the required page is not in the buffer is given as $(1 - b / (p_d' + 1))$. The required page can be any of the p_d' data pages to be accessed or the current index page. As the number of available page in the buffer approaches zero, the amount of paging becomes $2n$, as requests alternate from index page to data page and back. On the other hand, as the buffer size approaches $p_d' + 1$ so too does the amount of paging.

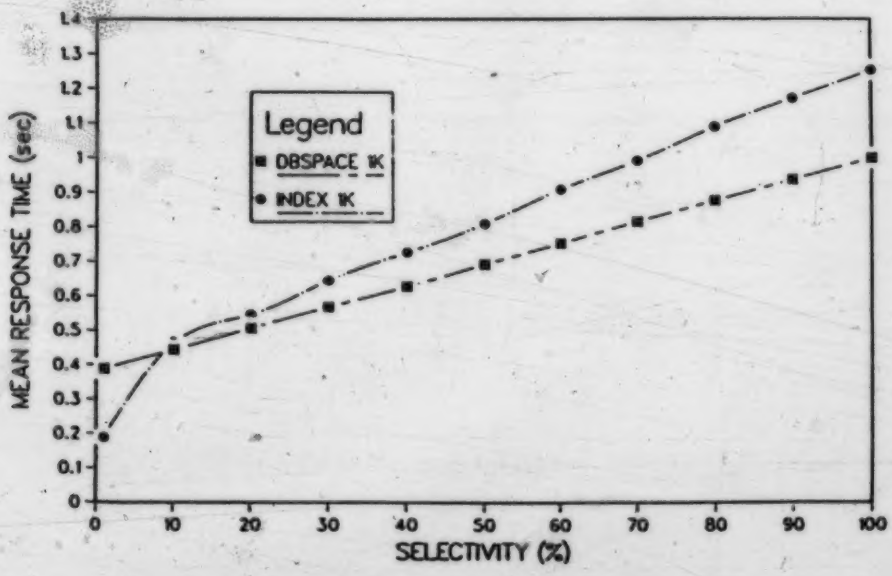

Figure 3.1 Query Type 1, Effect of Selectivity on Response Time

Figure 3.1 shows the effect of selectivity on response time for a relation of cardinality 1000 or 1K (which we will refer to, in this chapter, as the "small relation"). When referring to the number of tuples in a

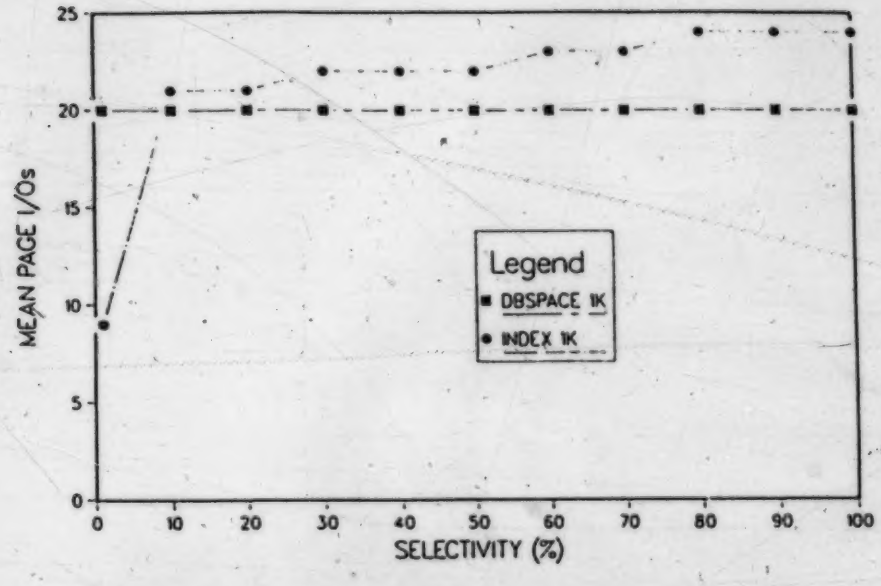

Figure 3.2 Query Type 1, Effect of Selectivity on Paging

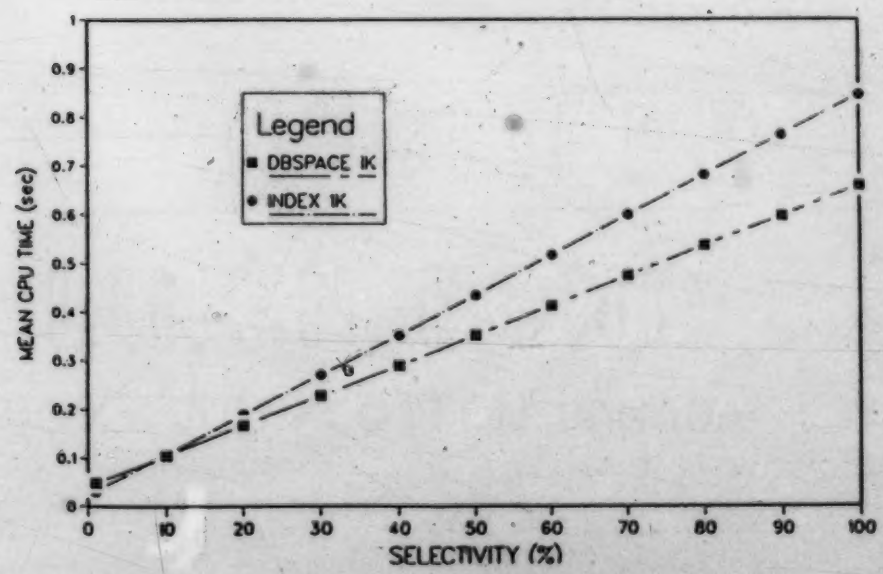

Figure 3.3 Query Type 1, Effect of Selectivity on CPU Time

relation, K is understood to mean 1000 tuples as opposed to 1024. At 50 tuples per page, the relation occupies 20 pages. Index pages contain 250 entries and, as such, the index occupies 4 pages. Buffer size has been set at the default value of 40 pages each of which is 4K in size. The trends for I/O activity are presented in Figure 3.2 while the mean CPU times are shown in Figure 3.3. Actual values and confidence intervals for these graphs are presented in Appendix 1. There is a strong relationship between paging and response time for this type of query. Although CPU time curves resemble response time curves to a certain degree, we shall see later that this does not always indicate that the CPU is the determining factor. The relationship between response time and paging will, however, be maintained.

The initial sharp increase in the index scan response time, in Figure 3.1 is primarily the result of increased paging. The point at which the response time exceeds that of the dbspace scan is approximately where each page is accessed exactly once. Since the buffer is large enough to contain all the pages, the increase after this point is mostly due to an increase in the number of tuples retrieved (i.e. increased selectivity). The slope of the curve for the index scan is slightly higher than that of the dbspace scan, because, as the selectivity increases so does the number of index pages that must be retrieved.

If the buffer space is limited, the response time of the index scan would increase faster because the page faults resulting from the random probing would require additional I/O. Our results indicate that the dbspace scan is totally unaffected by the varying buffer size since it accesses each page only once and can, therefore, easily buffer them one at a time.

The behavior of the larger relation of cardinality 5000 (which we will refer to as the "large relation") exhibits similar trends to that of the smaller relation. Thus, above a certain size of relation, the observations with regard to the buffer size and selectivity hold regardless of the relation cardinality.

3.1.2. Effect of Cluster Degradation

The use of a clustered index will always outperform a dbspace scan, unless the selectivity is such as to guarantee that almost all the pages will be accessed. Note that the level of selectivity that would guarantee the accessing of each page is quite different for the clustered and the non-clustered indices. If the tuples are stored in physical proximity to one another, as in the case of clustered indices, the selectivity has to be very high in order to access all the pages. The degradation of a clustered index, at high levels of selectivity, provides no additional information since the clustered index scan is more costly to begin with and the degradation of the clustering will result in random probing driving its cost even higher.

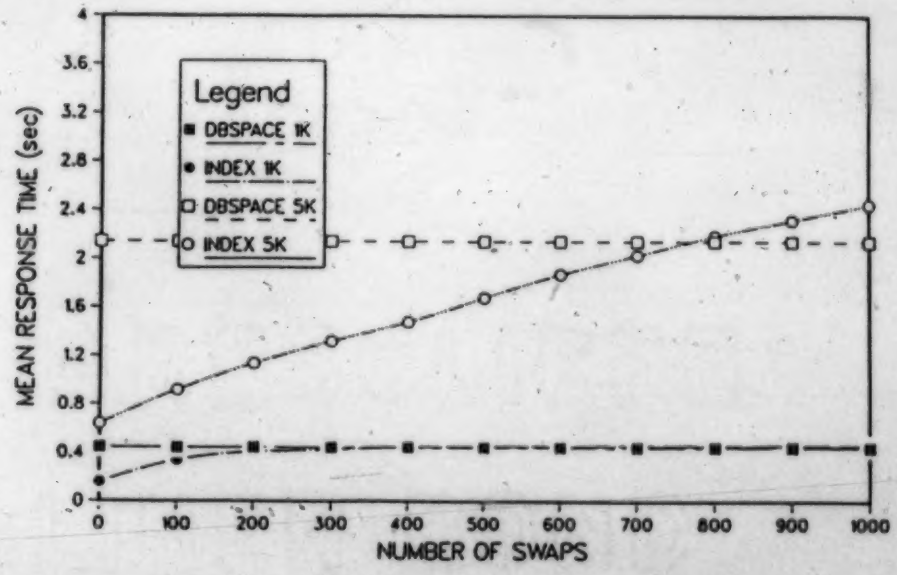

Figure 3.4 Query Type 1, Effect of Cluster Degradation on Response Time

At low levels of selectivity such that the number of tuples retrieved is less than the number of pages, the degradation of the clustering again offers little information. This is due to the fact that the index scan still outperforms a dbspace scan even in the worst case where one page is read in for each retrieved tuple. This result is independent of the buffer size.

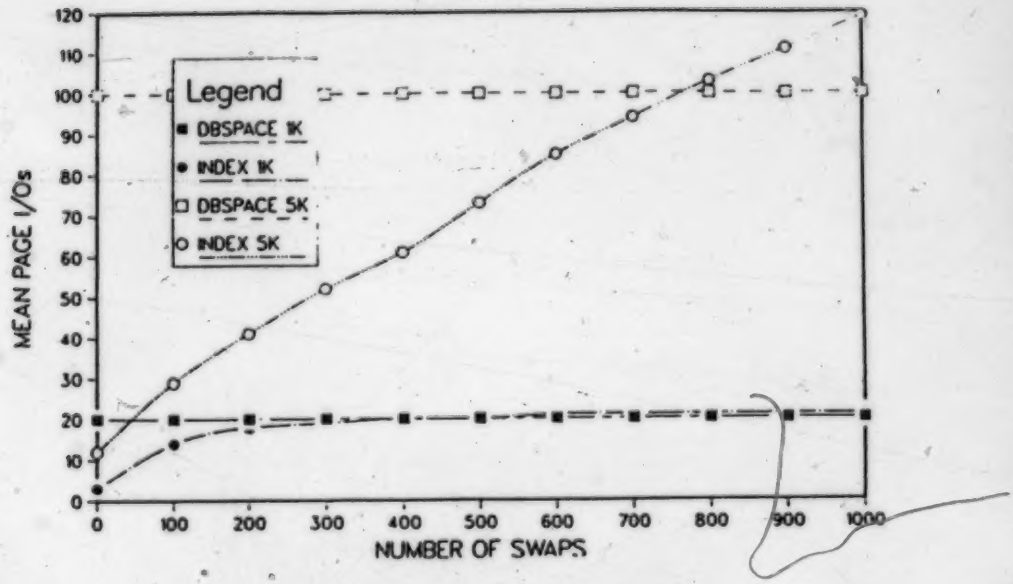

Figure 3.5 Query Type 1, Effect of Cluster Degradation on Paging

In between these two extremes, the degree of degradation to a clustered index will determine the optimal access path. If the relation is completely clustered, the number of data pages touched can be approximated by $p_d' = \lceil n / t_p \rceil$ where t_p is the number of tuples per page. This is true, because each page will be accessed only once and p_d' is the number of data pages that is required to hold n tuples. The number of data pages touched after degradation of the clustering by υ updates can be approximated by

$$p_d'' = p_d' + (p_d - p_d')(1 - ((p_d - p_d') / p_d)^{\upsilon}) \qquad \text{(Eq. 3.2).}$$

The term $((p_d - p_d') / p_d)^{\upsilon}$ is the probability that a given page of the remaining $p_d - p_d'$ pages does not contain a required tuple after υ updates. If $p_d' = p_d$ then the required number of pages remains at p_d independent of υ. If there are no updates (i.e. $\upsilon = 0$) then the required pages remain at p_d'. As υ increases, p_d'' will tend toward p_d.

Figures 3.4 and 3.5 are the cases where buffer size is 40 pages and the selectivity is 10% for both a small and a large relation. Figure 3.4 shows the effect of cluster degradation on response time while Figure 3.5 shows the effect on paging. The CPU times remain unchanged since the number of tuples retrieved

remains constant. For both the small and large relation, paging is closely related to response time.

Since the small relation will fit completely into the buffer, the increase in response time for an index scan is based solely on an increase in the number of pages containing requested tuples. No random probing occurs and an upper bound of the number of data pages and index pages is quickly reached.

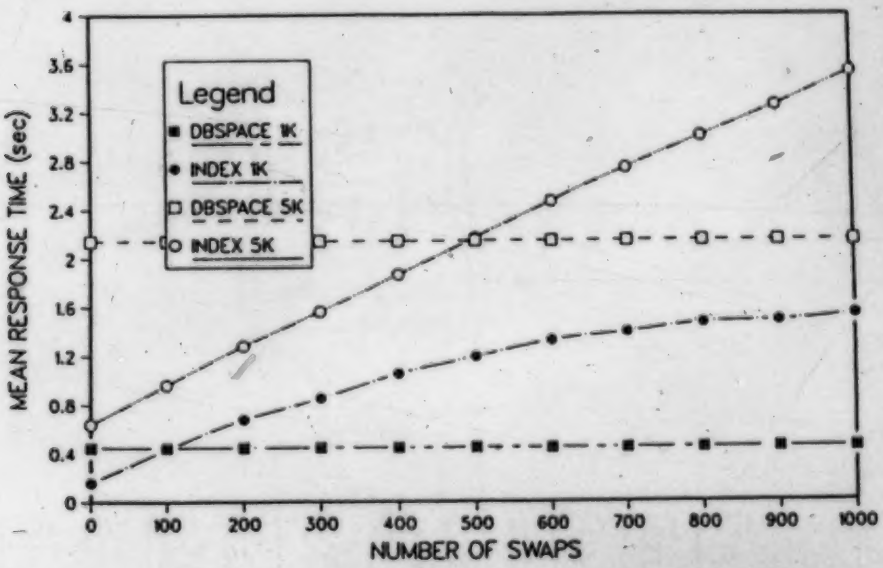

Figure 3.6 Query Type 1, Limited Buffer Space, Effect of Cluster Degradation

For the large relation which cannot fit entirely into the buffer, the upper bound on data pages requested is equal to the number of tuples retrieved. This is because, in the worst case of random probing, the next page required is never in the buffer. By limiting the buffer size for a small relation, the trends exhibited for the small relation becomes similar to that of the large relation. This is shown is Figure 3.6 where the buffer size has been reduced to 4 pages.

3.1.3. Summary

Based on these results one may conclude the following heuristics for type 1 queries:

```
If no index is available then
    use a dbspace scan
Else If the index is completely clustered then
    If ⌈n / t_p⌉ + p_i < p_d then
        use an index scan
    Else
        use a dbspace scan
    Endif
Else If the index is not clustered then
    If n + p_i < p_d then
        use an index scan
    Else If p_d > p_d' + p_i then
        If p_d > b + (2n − b)(1 − b / (p_d' + 1)) then
            use an index scan
        Else
            use a dbspace scan
        Endif
    Else
        use a dbspace scan
    Endif
Else If the clustered index is degraded then
    If n + p_i < p_d then
        use an index scan
    Else If p_d < ⌈n / t_p⌉ + p_i then
        use a dbspace scan
    Else If p_d > p_d'' + p_i then
        If p_d > b + (2n − b)(1 − (b / p_d'')) then
            use an index scan
        Else
            use a dbspace scan
        Endif
    Else
        use a dbspace scan
    Endif
Endif
```

3.2. Evaluation of Type 2 Queries

Type 2 queries are more complicated to execute because an ordering is requested on the attribute according to which the selection is performed. There are three ways to accomplish this:

(1) Retrieve tuples using an index on the attribute A_i.

(2) Perform a dbspace scan applying the predicate and then sort the resulting relation.

(3) Sort the entire relation and then perform a dbspace scan.

In Section 3.2.1 we report the results of the experiments with buffer size and selectivity. The effects of cluster degradation of indices in discussed in Section 3.2.2. In Section 3.2.3 a summary of the conclusions are given and some potential improvement are discussed.

3.2.1. Effect of Buffer Size and Selectivity

Figures 3.7 to 3.9 show the effect of selectivity on response time, paging, and CPU time, respectively, for type 2 queries when buffer size is fixed at 40 pages. Actual values and confidence intervals are provided in Appendix 1. The insensitivity of response time to the CPU processing time is much more evident, as is the dependence on paging.

If the predicate is being applied after the sort, the relation is loaded directly into the sort routine. A temporary relation is generated in sorted order and a dbspace scan is then applied. If the predicate is being applied before the sort, a dbspace scan is performed generating a temporary relation of targeted tuples. This relation is then loaded into the sort routine generating a second temporary relation on which a dbspace retrieves all tuples. As selectivity increases, more tuples are included in the sort. The addition of 1 tuple doesn't contribute that much to response time. Once the additional tuples contribute an additional page, the contribution becomes quite large. This is evident by the trend exhibited when applying the predicate before the sort.

In general, applying the predicate before the sort is preferred. The cost of creating an additional temporary relation is outweighted by a reduction in sorting. However, as selectivity increases and the cost of

sorting approaches the cost of sorting the entire relation, the additional temporary relation becomes redundant. This phenomenon, which occurs in the area of 70 to 75% selectivity is clearly seen in Figure 3.7 for both the small and large relations.

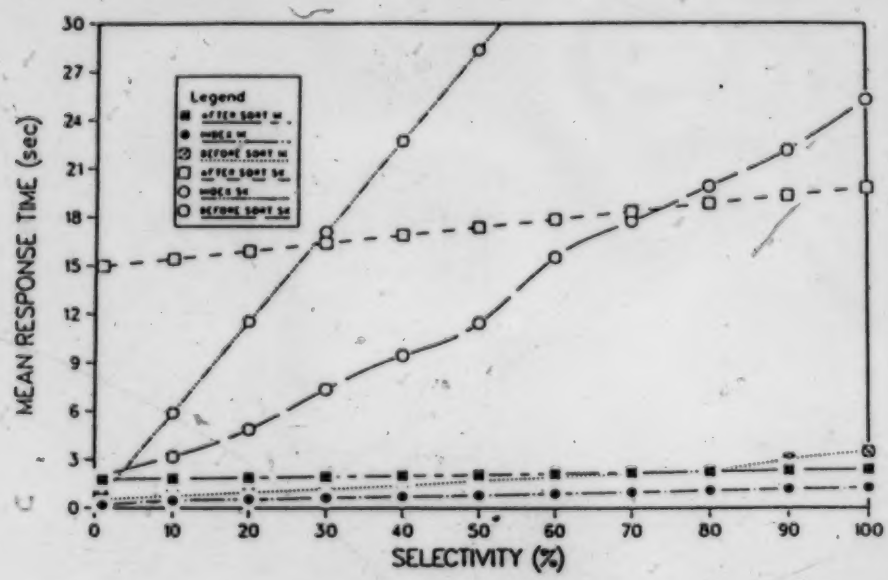

Figure 3.7 Query Type 2, Effect of Selectivity on Response Time

Recall from the previous section that when selectivity is sufficiently low, an index scan will perform better than a dbspace scan. Since the sort scans are based on dbspace scans, one would expect the relative performance of an index scan to be similar. In fact, if enough buffer space is available to hold the entire relation plus one additional page for the current index page, then an index scan is always preferred.

The ample buffer space removes the phenomenon of random probing. As in the previous section, when selectivity is low, the index scan simply touches fewer pages than a dbspace scan. Since the sort scan is based on the dbspace scan, the index scan offer better response time here as well. As the selectivity increases, the index scan will, eventually, touch all pages. The sort scans, in addition to touching all data pages to begin with, will be sorting temporary relations as well. Therefore, if there is ample buffer space, an index scan will always be preferred.

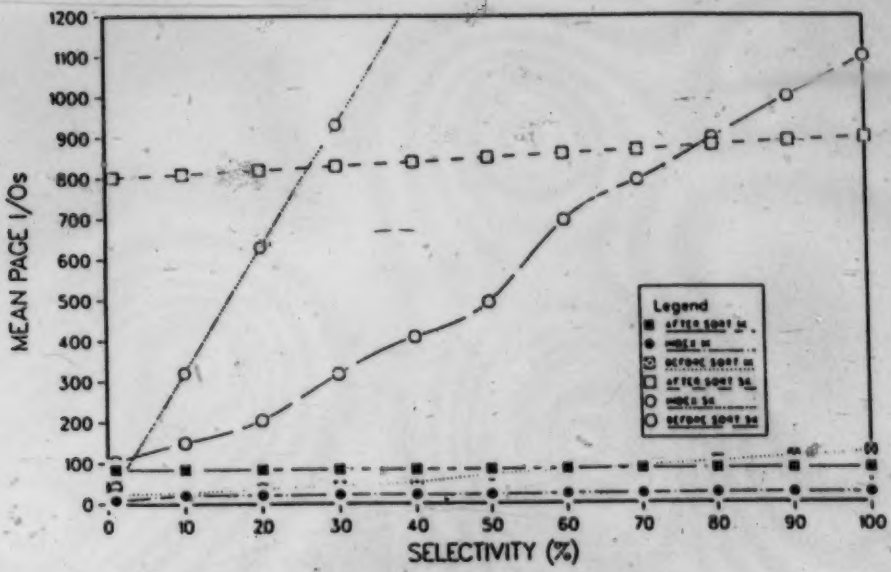

Figure 3.8 Query Type 2, Effect of Selectivity on Paging

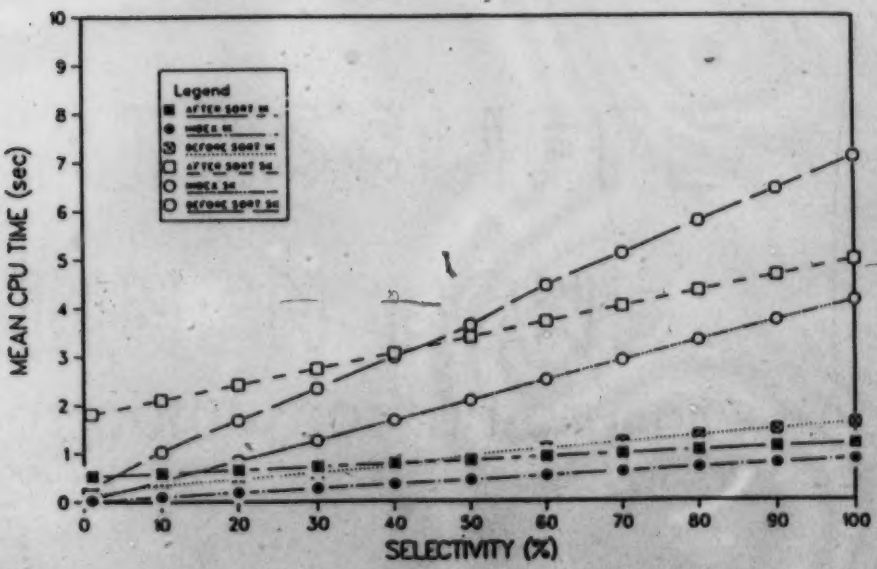

Figure 3.9 Query Type 2, Effect of Selectivity on CPU Time

If buffer space is limited then random probing is introduced and the trends indicated are similar to those of the previous section. Note that at low levels of selectivity, the response time for applying the predicate before the sort is similar to the dbspace scan. The resulting temporary relation to be sorted is relatively small and the cost of sorting is quite small as well. As selectivity increases, it is possible that each new tuple retrieved during an index scan causes a page access. With the sort, many additional tuples may be required to cause additional paging. As such, the increase in response time for the index scan is sharper and more steady than the sort scan.

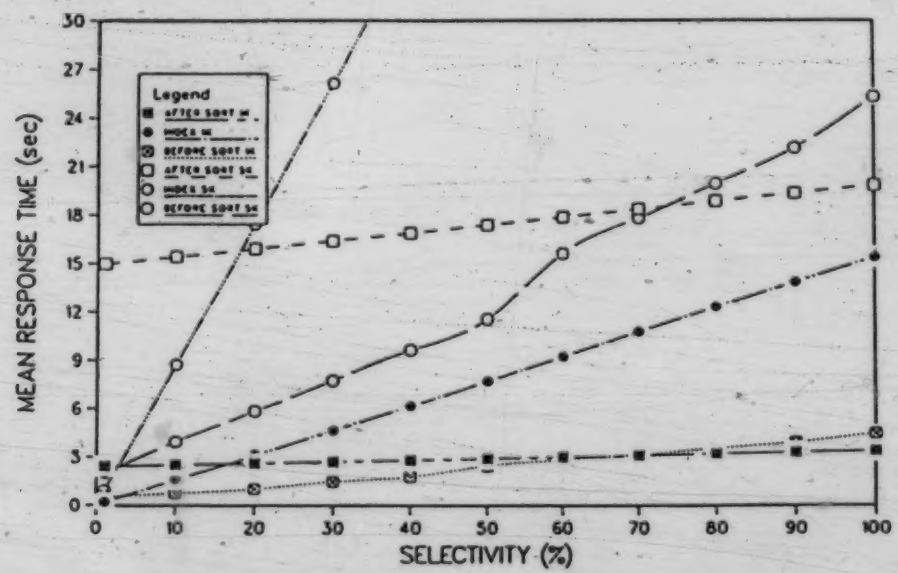

Figure 3.10 Query Type 2, Limited Buffer Space, Effect of Selectivity

By limiting the buffer size for the small relation we would expect the trends of the 3 approaches to become similar to those of the large relation. Figure 3.10, where the buffer size is reduced to 4 pages, shows that this is true.

3.2.2. Effect of Cluster Degradation

In the previous section we had stated that at low levels of selectivity the degradation of the clustered index has little effect, and an index scan will always perform better than a dbspace scan in this case. Since sort scans are inherently more costly, the same can be expected to hold true for Type 2 queries. In fact the range within which the index scan performs better is extended simply because sort scans are considerably more expensive than dbspace scans. For example, Figure 3.11 shows the response time for the case where the buffer size is 40 pages and the selectivity is 50% for both the small and the large relation. As expected, the small relation, with ample buffer space, always dictates that an index scan should be used independent of tuple degradation. For even the large relation, a substantial degradation is required to render an index scan non-optimal. In fact, an index scan is preferred until degradation has become substantial. Figure 3.12 gives the effect on paging, where again, the strong relationship with paging is evident.

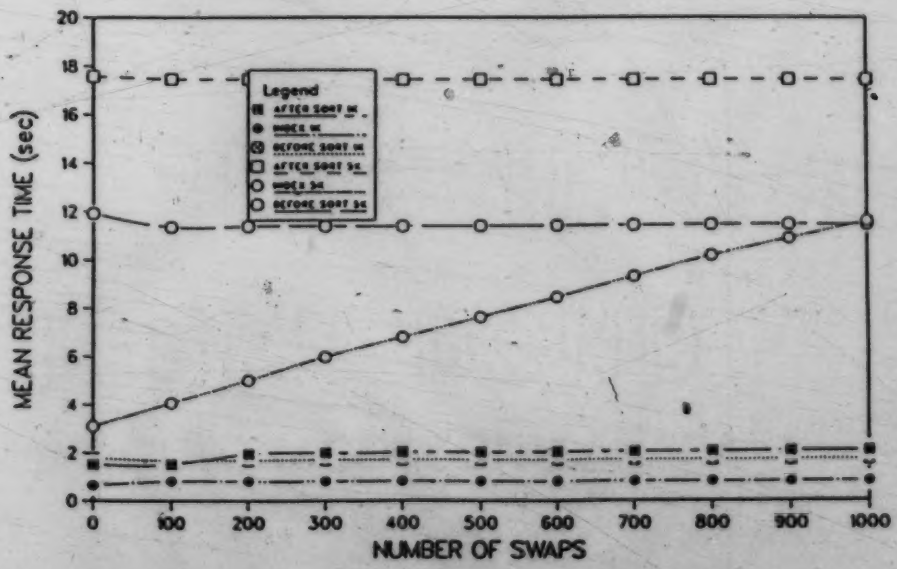

Figure 3.11 Query Type 2, Effect of Cluster Degradation on Response Time

Note (Figure 3.11) the sudden increase in the response time, after 100 SWAPS of the sort scan when the predicate is applied following the sort for a small relation. This is a result of page faults incurred during

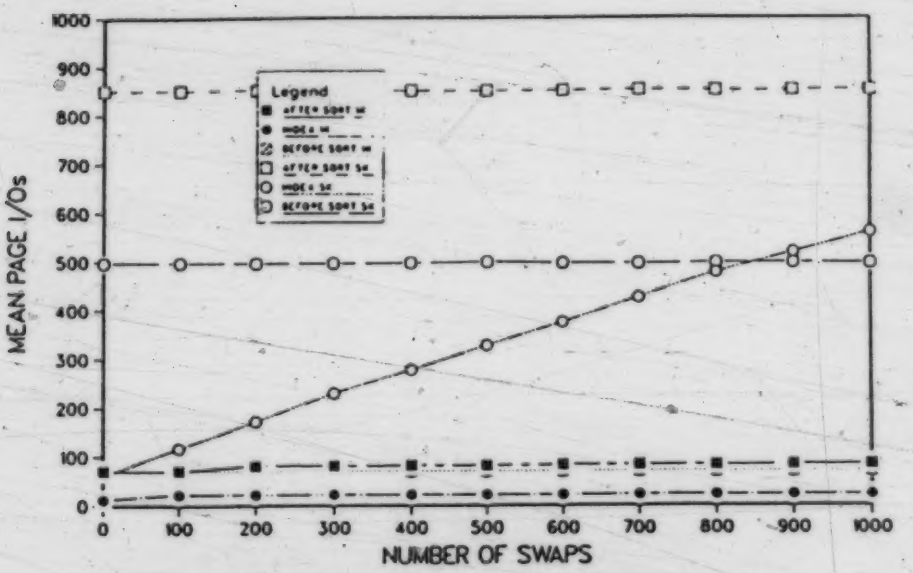

Figure 3.12 Query Type 2, Effect of Cluster Degradation on Paging

the merging phase of the sort. When clustering degrades, the merge phase of the sort algorithm will start reading tuples from a number of pages instead of one page. This will generate a reference pattern that will cause the LRU buffer management algorithm to swap out some pages that will be accessed immediately in the next phase, causing unnecessary page faults. In our experiments, this phenomenon becomes apparent when 400 tuples (200 SWAPs) get out of their clustering order. This is quite a complex phenomenon and requires a detailed description.

Assume that the relation to be sorted occupies 16 pages. Since the algorithm performs 8-way merges, the first 8 pages (p1 to p8) are loaded into 8 buffer frames (say, F1 to F8 in Figure 3.13a). Note that the numbers at the bottom indicate the order in which the system accesses the pages. Thus, page p1 is accessed and loaded first (i.e. frame F1 is the least recently used) and p8 is accessed and loaded last (i.e. frame F8 is the most recently used). Finally, note that since the pages are clustered, they are also sorted among themselves. In other words, the largest sort attribute value in page p1 is smaller than the smallest sort attribute value in p2, and this is true for all pages.

Since there are 16 pages and the merge-sort algorithm, as we modeled it, performs 8-way merges, the merging phase of sorting the relation will require 3 passes. These 3 passes are one to merge the first 8 pages, one to merge the next 8 pages, and the last pass to merge the two 8 page groups.

To explain the behavior demonstrated in Figure 3.11, assume that each page is sorted within itself. If the clustering is maintained, then the merge-sort algorithm will copy all the tuples in F1 into F9 followed by coping all the tuples of F2 into F10. This will generate a reference pattern as depicted in Figure 3.13b. Notice that F1 is still the least recently used page with F16 being the most recently used.

In the second pass, pages p9 to p16 are loaded into frames F17 to F25 (Figure 3.13c). Since these pages are also clustered and in sorted order, the merge-sort algorithm behaves similar to the first pass, resulting in a reference pattern depicted in Figure 3.13d.

In the third pass, we have only two groups to merge, and therefore, require only a 2-way merge. The algorithm reads one tuple form page p1 in frame F9 and one tuple from page p9 in frame F25. The comparison of the sort attribute will indicate that the tuple from p1 should appear before the tuple from p9 in the sorting order. This will be true for every tuple of p1. Therefore, the algorithm will attempt to copy (one by one) the tuples from frame F9 to an empty frame. Since there are no empty frames, the LRU buffer management will swap out the least recently used frame, which is F1. The merging will continue in this uniform fashion and result in the sorted relation to be stored in frames F1 to F8 and F17 to F24, in that order (Figure 3.13e).

Note that if clustering is maintained, the sorting continues without causing any unnecessary page faults. If, however, clustering is degraded, then the pages will not stay in sorted order relative to one another. Therefore, merging will require tuples to be read from a number of frames, generating a reference pattern that is not uniform. This will result in the swapping out of some pages that will be accessed in the next phase of the merge-sort algorithm.

By limiting the buffer space as in Figure 3.14, in which the buffer size has been reduced to four pages, the sudden increase doesn't occur. At this level of selectivity with such limited buffer space the

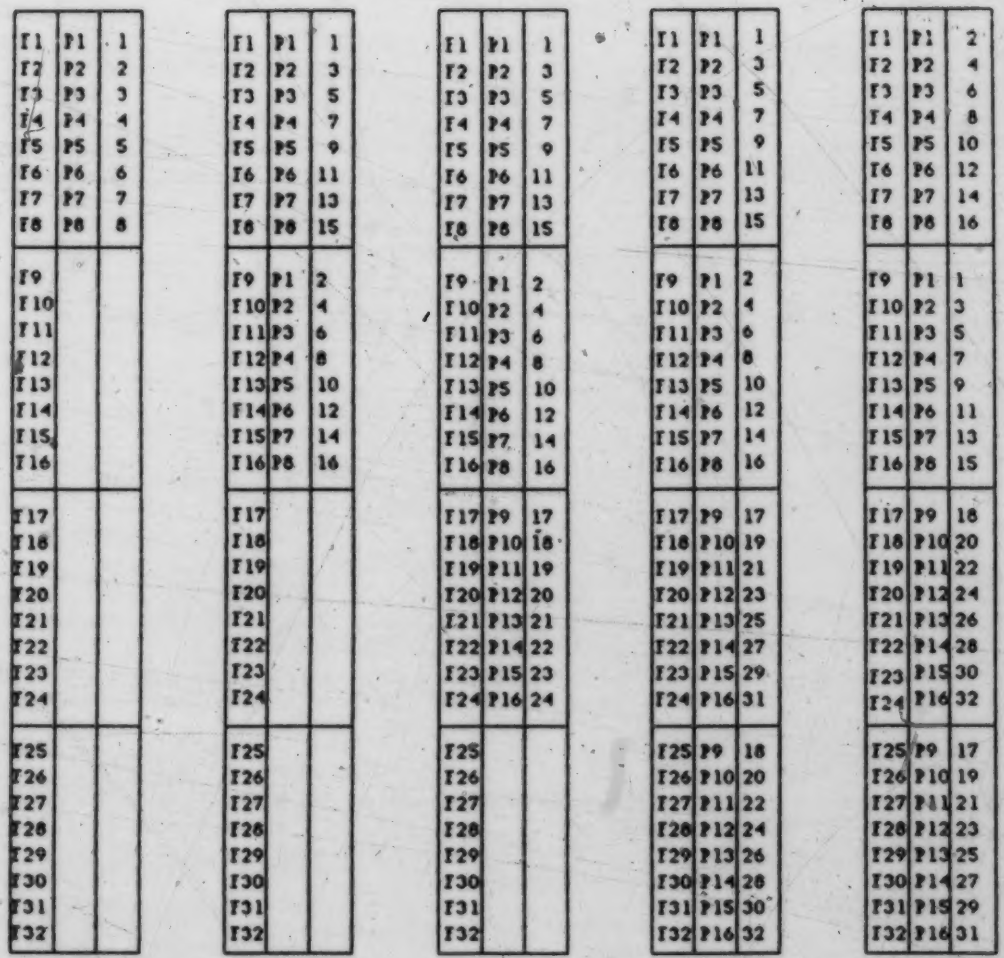

Figure 3.13 Sort-Merge Buffering

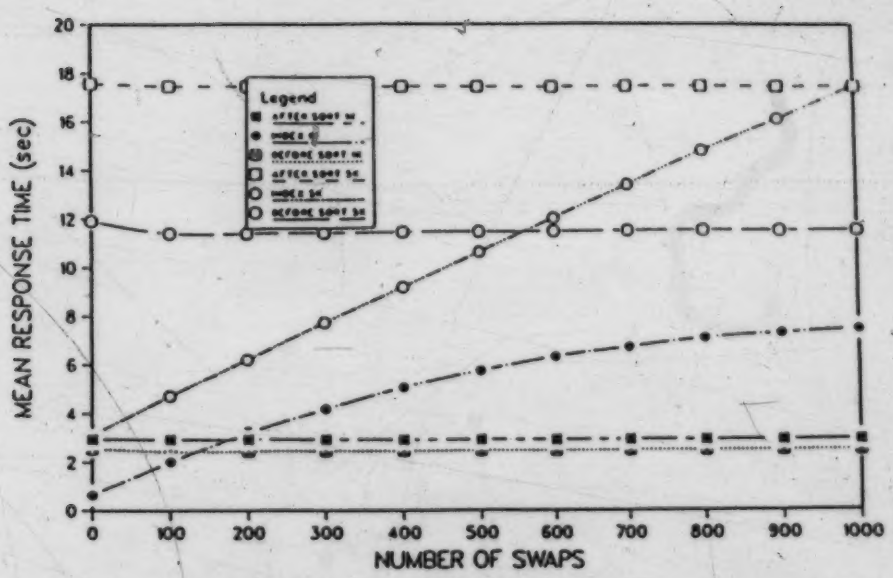

Figure 3.14 Query Type 2, Limited Buffer Space, Effect of Cluster Degradation

index scan on the small relation becomes non-optimal very quickly.

3.2.3. Summary

Based on these results one may conclude the following heuristics for type 2 queries:

If no index is available then
    If selectivity > 75% then
        apply the predicate after the sort
    Else
        apply the predicate before the sort
    Endif
Else If the index is completely clustered then
    use an index scan
Else If the index is not clustered then
    If $b > p_d$ then
        use an index scan
    Else If $n + p_i < p_d$ then
        use an index scan
    Else If $p_d > p_d' + p_i$ then
        If $p_d > b + (2n - b)(1 - b / (p_d' + 1))$ then
            use an index scan
        Else If selectivity > 75% then
            apply the predicate after the sort
        Else

```
            apply the predicate before the sort
        Endif
    Else If selectivity > 75% then
        apply the predicate after the sort
    Else
        apply the predicate before the sort
    Endif
Else If the clustered index is degraded then
    If b > p_d then
        use an index scan
    Else If n + p_i < p_d then
        use an index scan
    Else If p_d < ⌈n / t_p⌉ + p_i then
        If selectivity > 75% then
            apply the predicate after the sort
        Else
            apply the predicate before the sort
        Endif
    Else If p_d > p_d'' + p_i then
        If p_d > b + (2n − b)(1 − (b / p_d'')) then
            use an index scan
        Else
            If selectivity > 75% then
                apply the predicate after the sort
            Else
                apply the predicate before the sort
            Endif
        Endif
    Else
        If selectivity > 75% then
            apply the predicate after the sort
        Else
            apply the predicate before the sort
        Endif
    Endif
Endif
```

The sort used here does not make very good use of buffer space and the need for sort algorithms taking advantage of the pages available should be obvious. In addition, a buffer management system sensitive to the needs of query processing is required. The incorporation of the hot-set model may be attractive.

3.3. Evaluation of Type 3 Queries

The results of the Type 3 queries must be in sorted order on an attribute (A_j) different than the one that participates in the selection predicate (A_i). There are four ways to accomplish this:

(1) Retrieve tuples using an index on the attribute A_i and then sort the result on A_j.

(2) Access the tuples using an index on the attribute A_j and perform the selection according to A_i on each accessed tuple.

(3) Perform a dbspace scan applying the predicate on A_i and then sort the resulting relation on A_j.

(4) Sort the entire relation on A_j and then perform a dbspace scan applying the predicate on A_i.

In Sections 3.3.1 we report the results of the experiments with buffer size and selectivity. The effects of index clustering on the behavior of this type of queries is discussed in Section 3.3.2. Section 3.3.3 presents a summary of the conclusions drawn.

3.3.1. Effect of Buffer Size and Selectivity

When the tuples of the relation are accessed according to the index on A_j, alternative (2) discussed above, the resulting relation will be in sorted order. However, independent of the level of selectivity, every page in the relation will be retrieved. This is because the predicate must be applied against the tuples and cannot be applied against the index to restrict the search. If ample buffer space is available such that the entire relation can be accommodated, the amount of paging is restricted to the number of pages occupied by the relation and the index. Figure 3.15, which depicts the response time results for the small relation with a buffer size of 40 pages, clearly shows that for all but extremely low levels of selectivity it is preferred to use an index on the sorting attribute and then apply the selection predicate. Figures 3.16 and 3.17 show the effect on paging and CPU time respectively. Actual values and confidence intervals are given in Appendix 1.

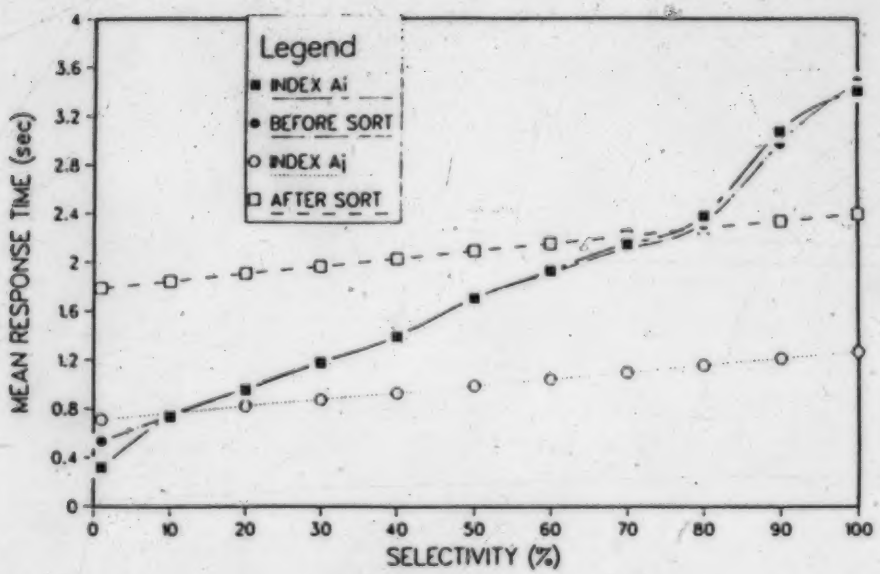

Figure 3.15 Query Type 3, Effect of Selectivity on Response Time

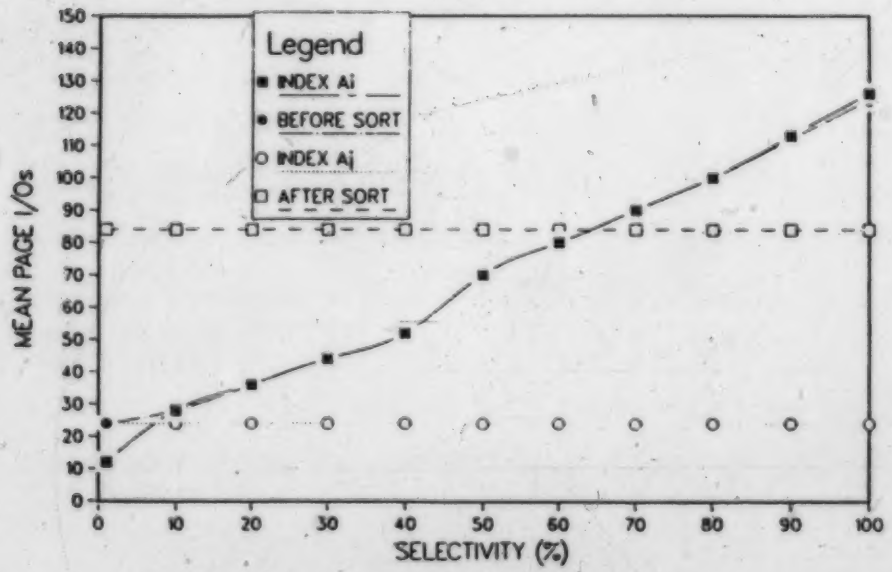

Figure 3.16 Query Type 3, Effect of Selectivity on Paging

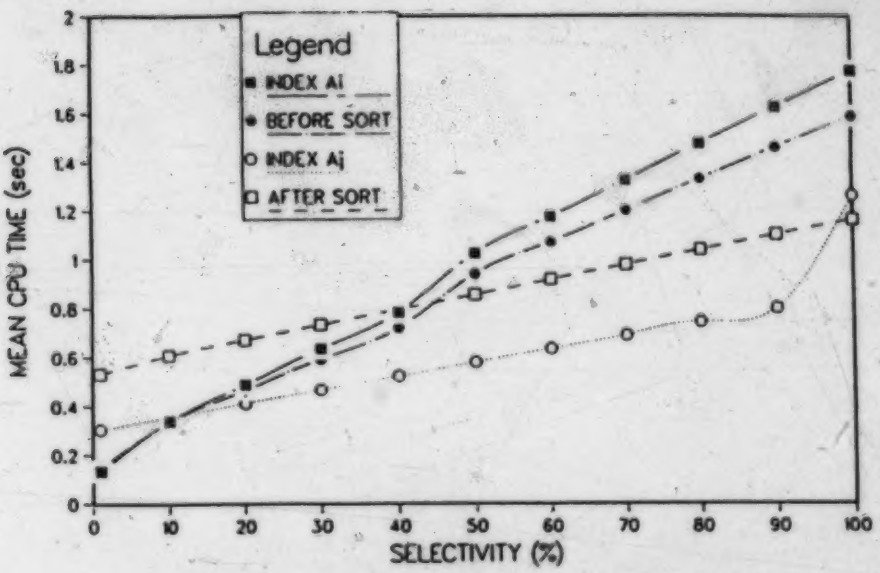

Figure 3.17 Query Type 3, Effect of Selectivity on CPU Time

The slight increase in response time of the index scan on A_j is associated solely with the number of tuples retrieved and is not a result of additional paging. Similarly, when applying the predicate after the sort, all pages must be retrieved and sorted independent of the selectivity. The increase that parallels that of the index scan on A_j is again a result of the number of tuples retrieved.

In the case where index scan is performed on A_i and the predicate is applied before the sort, the increase in selectivity results in increased paging. Therefore, the increase in response time is much more dramatic. Since there is sufficient buffer space available, each page of the relation will be retrieved only once. A similar trend is shown for the case where the predicate is applied before the sort. Each of these methods sort the same number of result tuples; however, when selectivity is low, the index scan on A_i retrieves fewer pages.

After selectivity has increased to the point where the index scan touches the same number of pages as the dbspace scan, the two approaches are equal. At high levels of selectivity the index scan is touching more pages than the dbspace scan and, as such, the response times are higher.

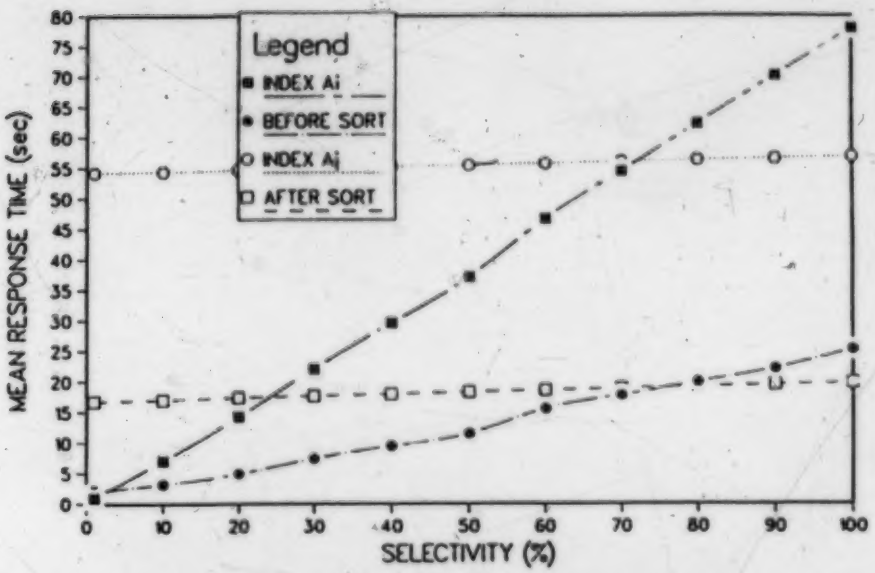

Figure 3.18 Query Type 3, Limited Buffer Space, Effect of Selectivity

When buffer space is limited and the relation is large, as in the case in Figure 3.18, the cost of random probing has eliminated the index scan on A_j as a reasonable alternative. Since all tuples are retrieved, this high level of paging is the same for all levels of selectivity. The performance of applying the predicate after the sort remains fundamentally unchanged. The index scan on A_i exhibits a dramatic increase in response time. Remember that when the buffer space is not limited, the increase in response time is only due to the increased sorting cost. With limited buffer space, however, there is the additional cost of page faults due to random probing. The relationship between applying the predicate before or after the sort is the same as that for the Type 2 queries.

3.3.2. Effect of Cluster Degradation

In the type of queries covered in the section we have been considering the possibility of two indices: one that may exist on the selection predicate attribute A_i and another that may exist on the ordering attribute A_j. A relation can be clustered with respect to only one index. Since we have shown previously in Section 3.1.2 that the degradation of clustering has little effect when ample buffer space is available, we restrict the discussion to the case when buffer space is limited.

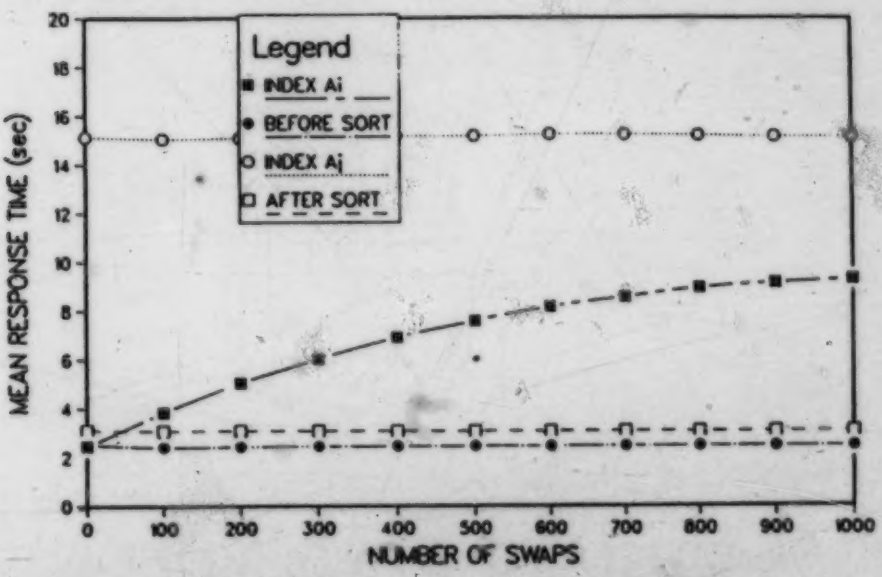

Figure 3.19 Query Type 3, Effect of A_i Cluster Degradation on Response Time

Figure 3.19 shows the response times for the case of a small relation with a buffer size of 4 pages and a selectivity of 50%. Figure 3.20 shows the effect on paging. Here we have clustered the relation with respect to the index on A_i. Figure 3.21 and 3.22 show a similar situation with clustering with respect to the index on A_j. In both cases it is evident that a small perturbation in the clustering quickly renders the index scan non-optimal.

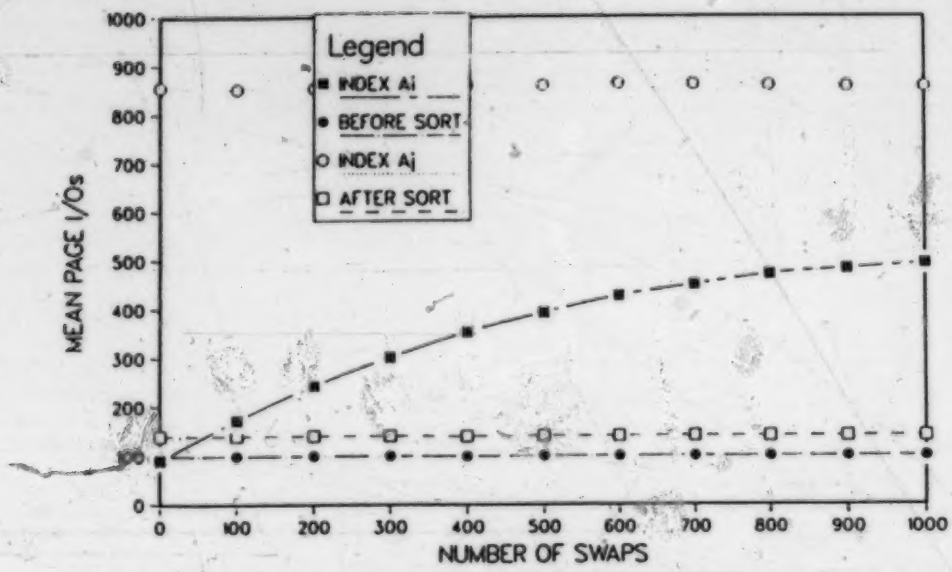

Figure 3.20 Query Type 3, Effect of A_i Cluster Degradation on Paging

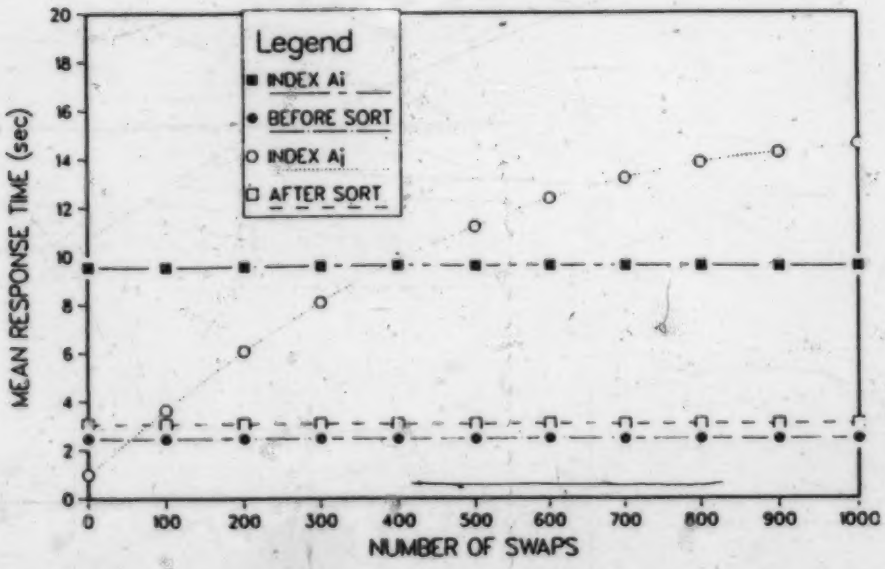

Figure 3.21 Query Type 3, Effect of A_j Cluster Degradation on Response Time

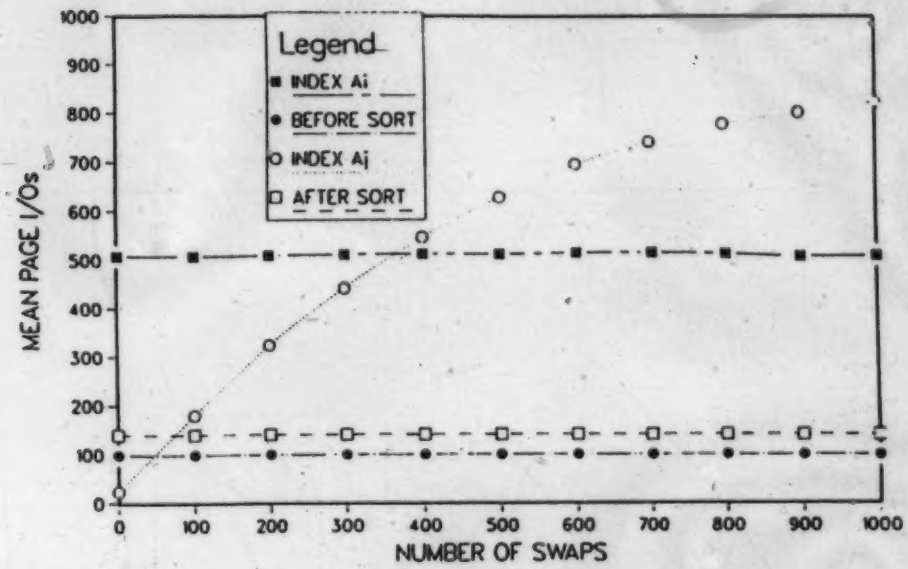

Figure 3.22 Query Type 3, Effect of A_j Cluster Degradation on Paging

3.3.3. Summary

Based on these results, one may conclude the following heuristics for type 3 queries:

If $n < p_d$ **then**
  If an index on A_i is available **then**
    use an index scan on A_i
  Else
    apply predicate before the sort
  Endif
Else If an index is on A_j **AND** $b > p_d$ **then**
  use an index scan on A_j
Else If selectivity < 75% **then**
  apply the predicate before the sort
Else
  apply the predicate after the sort
Endif

Chapter 4

JOIN EXECUTION

In this chapter we concentrate only on the joins. The joins that we consider are 2-way equi-joins, which is consistent with the current literature. An equi-join is a θ-join where θ is the equality operator. Assume that R is a relation defined over the set of attributes $A = \{A_1, A_2, ..., A_n\}$ and S is a relation defined over the set of attributes $B = \{B_1, B_2, ..., B_n\}$. A θ-join is defined as $\sigma_{A\theta B}(R \times S)$. That is, the equi-join of R and S is those tuples in the Cartesian product of R and S such that $A_i = B_j$.

The two most popular join methods are considered in this study: nested loop (NL) and merge scan (MS). A number of variations are available depending on the existence of indices on the join attribute. Table 4.1 presents a summary of these variations and gives the notation that we will be using in this chapter.

Join Method	Index		Sort		Notation
	Outer	Inner	Outer	Inner	
MS	Y	Y	N	N	MB
MS	N	N	Y	Y	MN
MS	Y	N	N	Y	MO
MS	N	Y	Y	N	MI
NL	Y	Y	N	N	NB
NL	N	N	N	N	NN
NL	Y	N	N	N	NO
NL	N	Y	N	N	NI

Table 4.1 Join Methods

Since we are concerned only with those methods providing the best response times, some of the approaches have been removed from further consideration. The results of the experiments for the nested loop methods with no indices (NN) and index on the outer relation (NO) are not presented. In both cases the response times have exceeded all others by several orders of magnitude. This is due to the manner in

which System R performs nested loop joins.

Recall that the interface between the RDS and RSS is performed by tuple-at-a-time operations on the base relations. This means that during a nested loop join, when no index is available on the inner relation, the entire inner relation must be sequentially scanned for every tuple of the outer relation. Even when the entire inner relation fits in the buffer, the CPU cost of opening a scan for every tuple of the outer relation is still quite large.

An alternative approach, nested block join [Sac86], provides far better performance, however this approach has not been implemented in System R. The nested block join simply performs a nested loop join by the block instead of by the tuple. All tuples in a block are applied to the inner relation on the same pass. For example, if a block is a page, then a scan is opened on the inner relation for every page in the outer relation.

Another approach that we have not considered in this study is the merge scan with an index on both relations (MB). This approach can never offer better performance than the nested loop scan with an index on both relations. Although the reference patterns are identical for both, the MB requires additional CPU time to maintain pointers on both relations [Loh87].

In this chapter we deal with two relation sizes: 1000 tuples (large relation) and 500 tuples (small relation). We have reduced the relation sizes in this chapter because the response times for joins over very large relations are very high and, as we shall see, results are dependent on the relation sizes relative to buffer size. In other words, we can achieve results similar to those that would be produced by larger relations by restricting the size of the buffer. The following sections, 4.1 to 4.2, deal with joins over two large relations and a small and a large relation, respectively. Each is investigated subject to join selectivity and buffer size. Join selectivity represents the proportion of tuples participating in the join. Section 4.3 presents a summary of the conclusions drawn.

Recall that the join selectivity is;

$$card(\sigma_{A.a8_i}(R \times S))/card(R) * card(S).$$

Therefore, the join selectivity is determined by the cardinality of the result relation. There is a variety of ways to achieve a given result size, depending on the relationship between the join attributes. Recall from Section 2.2.3.3 that the cardinality of the result relation can be given as $\alpha * (n * m)$. The relationship between the join attributes n and m may be one-to-one, one-to-many, or many-to-many. For example, 100 matches of 1-to-1 yield a result of 100 tuples, 10 matches of 1-to-10 yields a result of 100, and so too does 10 matches of 2-to-5. All give the same result relation size and, as such, the same join selectivity.

Each of the first two sections considers two ranges of join selectivity. The first, 0.00001 to 0.001, makes use of two ratios: 1-to-1 and 1-to-10. It was found that many to many offered little insight as the trends were similar to 1 to many at the same level of join selectivity. The second, 0.0001 to 0.01, makes use of one ratio: 10-to-10. The other two ratios can not be used to generate join selectivities in this range.

A one-to-one relationship implies that a given tuple in relation *A* matches one and only one tuple in relation *B*. In this case the join selectivity factor has an upper bound of *MIN* (*CardA* ,*CardB*) / (*CardA* * *CardB*) or 1000 / (1000 * 1000). That is, the highest join selectivity is 0.001.

A one-to-many relationship implies that a given tuple in relation *A* matches several tuples in relation *B*, but a given tuple in relation *B* matches only one tuple in relation *A*. An upper bound of *MAX* (*CardA* ,*CardB*) / (*CardA* * *CardB*) now holds, which, in our case is 1000 / (1000 * 1000) or a join selectivity is 0.001.

A many-to-many relationship implies that a given tuple in relation *A* matches many tuples in relation *B* and vise versa In this case the join selectivity factor has an upper bound of (*CardA* * *CardB*) / (*CardA* * *CardB*) or (1000 * 1000) / (1000 * 1000) = 1.

4.1. Joins Between Two Large Relations

In this section we consider the join between two large relations. Sections 4.1.1 to 4.1.3 present the results of experiments with attribute relationships of 1-to-1, 1-to-10, and 10-to-10, respectively. The first two provide a join selectivity factor ranging from 0.00001 to 0.001 while the last ranges from 0.0001 to 0.01.

4.1.1. One-to-One

Figures 4.1 to 4.3 show the effect of join selectivity on response time, paging, and CPU time, respectively, when buffer size is fixed at 40 pages. Actual values and confidence intervals are provided in Appendix 2.

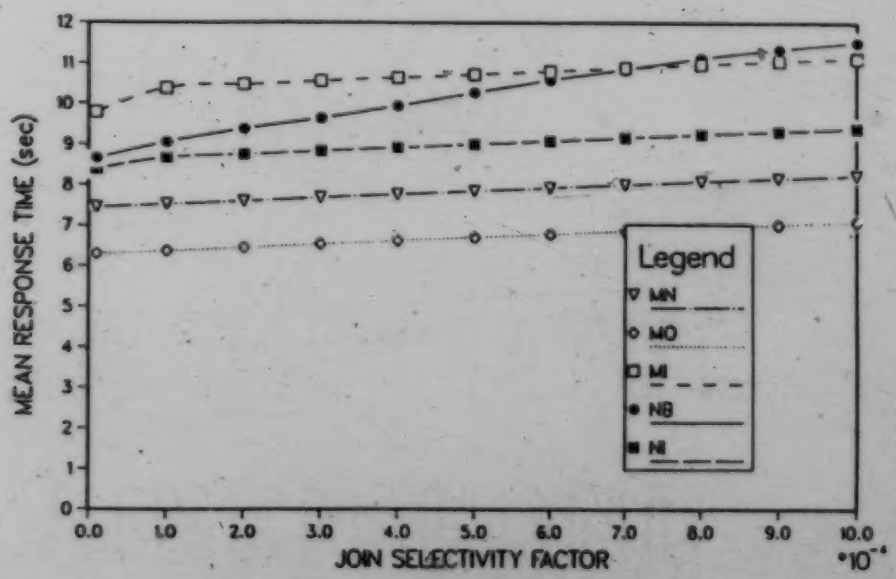

Figure 4.1 Effect of Join Selectivity on Response Time

For simple queries paging was the dominant factor, however, from Figures 4.2 and 4.3 it appears that CPU time has now assumed that role. The main culprit is the cost of opening a scan. For simple queries this is applied only once. When executing a join, scans are applied repeatedly, once for every outer relation

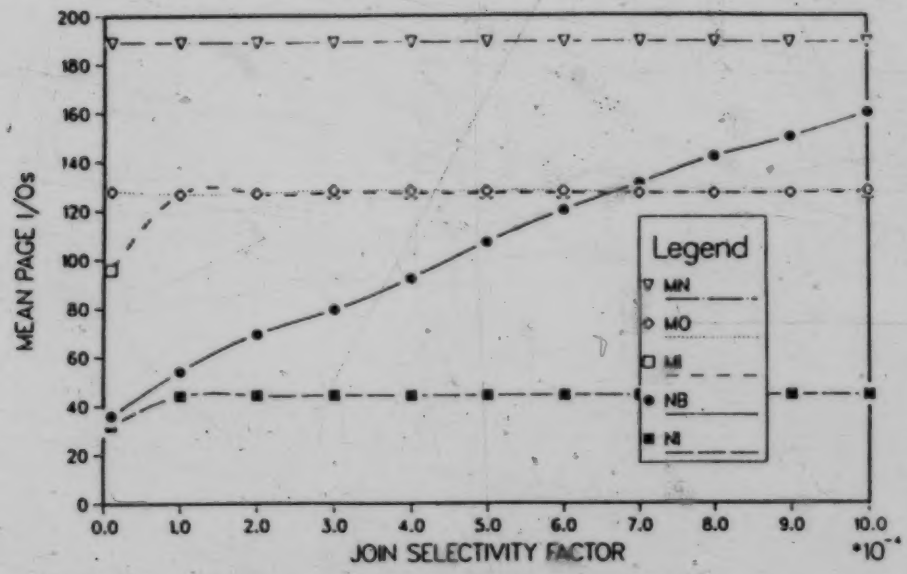

Figure 4.2 Effect of Join Selectivity on Paging

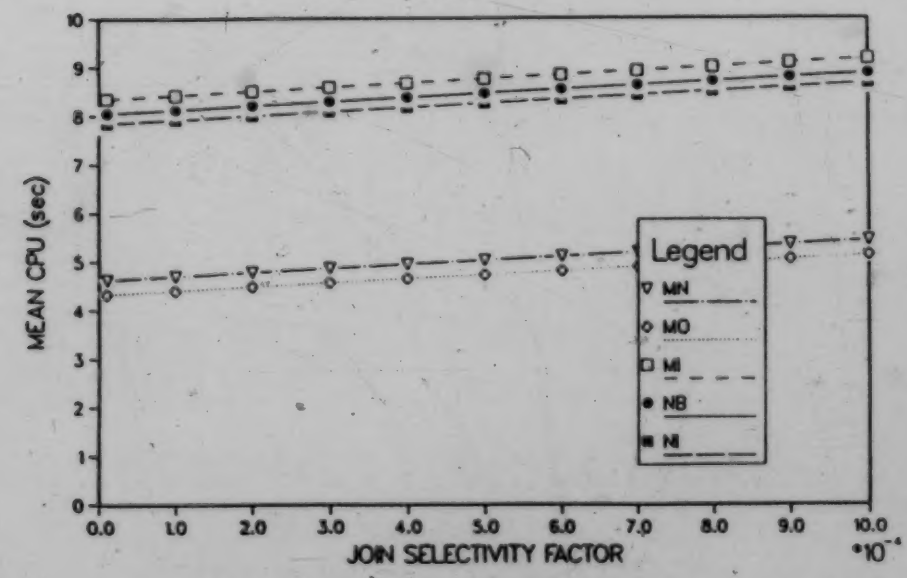

Figure 4.3 Effect of Join Selectivity on CPU Time

tuple.

From Figure 4.1, where we have fixed the buffer size to 40 pages making ample buffer space available, it is evident that if an index is available on the outer relation, a merge scan making use of the index is preferred (MO). With the inner relation sorted, no random probing occurs, minimizing the amount of buffer space required. Because of this, ample buffer space is available for the outer relation and sorting the outer relation becomes more expensive. If an index is not available then a merge scan that sorts both relations is preferred (MN), again due to the limited use of buffer space by the inner relation.

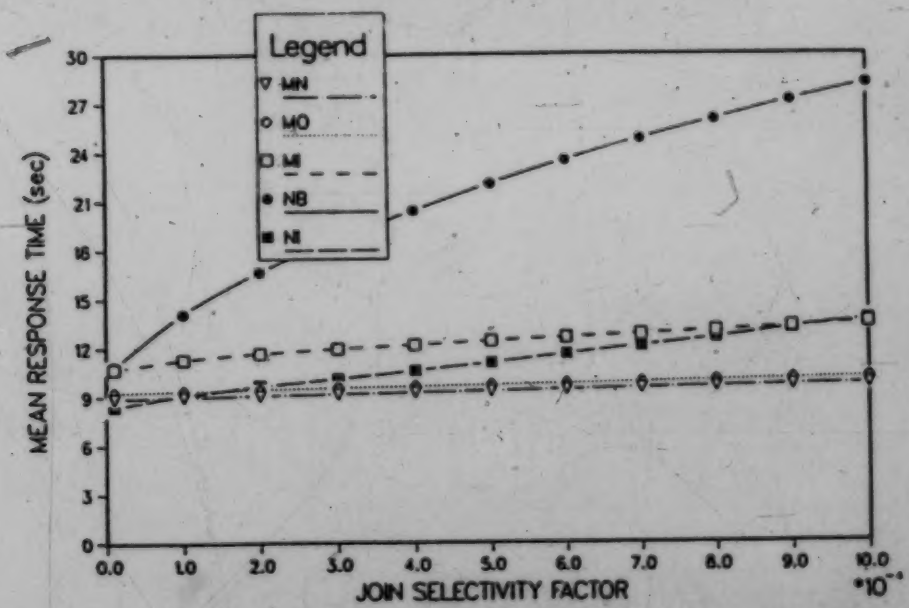

Figure 4.4 Effect of Join Selectivity on Response Time

In Figure 4.4 we have reduced the buffer size to 20 pages, large enough to hold only one of the relations. The limited buffer space causes random probing on the outer relation during the merge scan with an index on the outer and the extra effort to sort the relation become attractive.

At low levels of selectivity, a nested loop with an index on the inner relation is preferred. When selectivity is low, few tuples are retrieved. If the join attributes don't match, only the index page is accessed while random probing occurs for the outer relation.

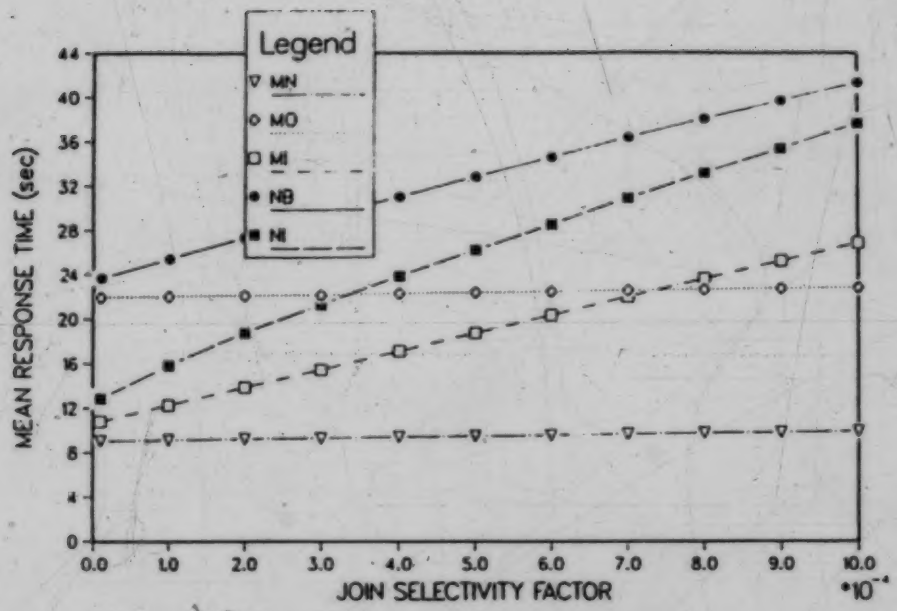

Figure 4.5 Effect of Join Selectivity on Response Time

Figures 4.5 and 4.6 show the case when buffer space is very limited. Now paging is very similar to response time. Recall that when selectivity is high and ample buffer space is available an index scan and a dbspace scan are similar. Each retrieves each data page once and only once. The same holds true for the scan of the outer relation for a join. Ample buffer space is available and selectivity is high since all tuples are being retrieved. Although the same does not hold for the inner relation, paging is kept at a minimum because the availability of buffer space prevents actions on the outer relation from interfering with paging on the inner. The net result is that paging contributes little in relative terms. Comparing these results with Figure 4.2, the largest change occurs for NB where the total paging increases only by approximately 120 pages which translates into approximately 2 seconds. The remaining approaches change little.

When buffer space is limited, as in Figure 4.6, paging becomes much more sensitive to selectivity and as such paging assumes a more dominant role. The degree of random probing has rendered any approach utilizing an index far too costly. The optimal approach is the merge scan sorting both relations (MN).

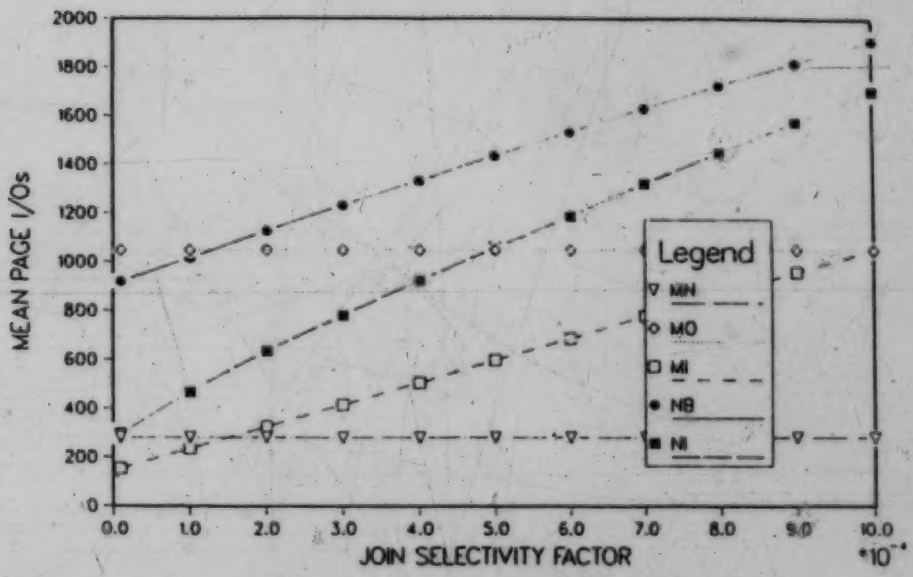

Figure 4.6 Effect of Join Selectivity on Paging

4.1.2. One-to-Many

Figures 4.7 to 4.9 show the effect of join selectivity on response time, paging, and CPU time, respectively, when buffer size is fixed at 40. Actual values and confidence intervals are provided in Appendix 2.

The NB and MI approaches offer higher response times. Since MN does not require the use of indices and offers superior response time, the presentation of those approaches within its range is sufficient. Only when an index is used on the inner relation does the location of the *many* component have an effect. The conclusions are similar to the previous section in that if an index is available on the outer relation then it is preferable to use a merge scan making use of the index (MO). If no index is available on the outer relation, then a merge scan that sorts both relations is preferable. Again note the correlation with CPU time. As buffer space becomes limited, the paging will again become dominant.

Figure 4.10 shows the effect of join selectivity on response time when buffer space has been restricted to 20 pages. Although similar conclusions can be drawn, note the effect of the location of the *many*

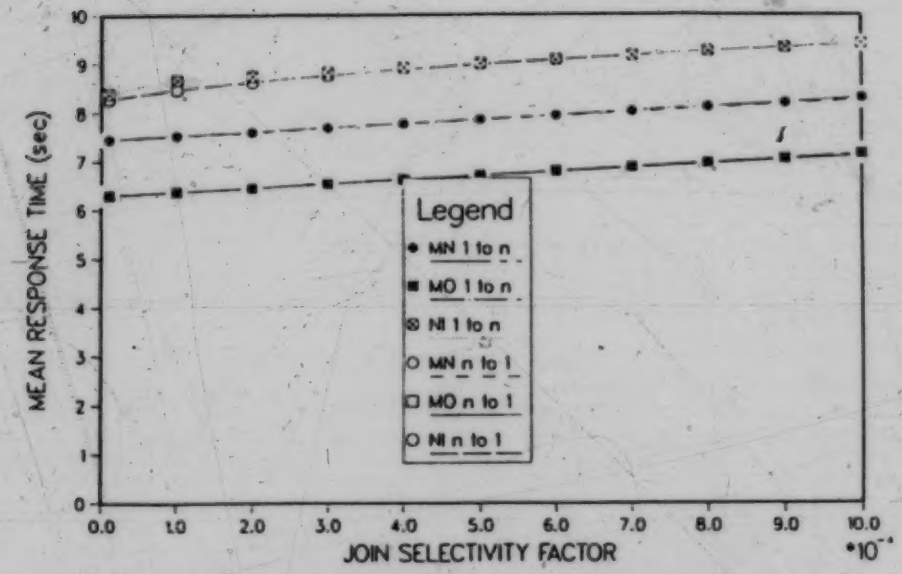

Figure 4.7 Effect of Join Selectivity on Response Time

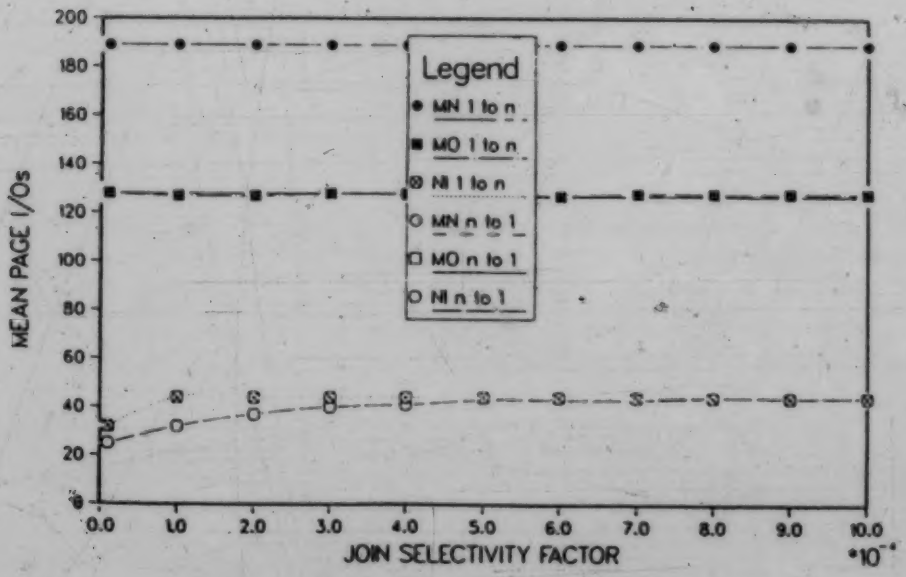

Figure 4.8 Effect of Join Selectivity on Paging

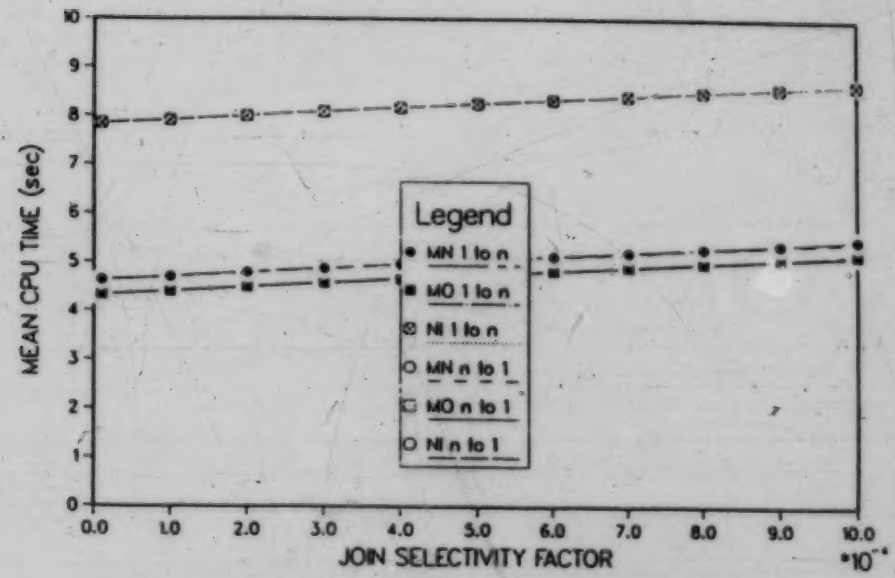

Figure 4.9 Effect of Join Selectivity on CPU Time

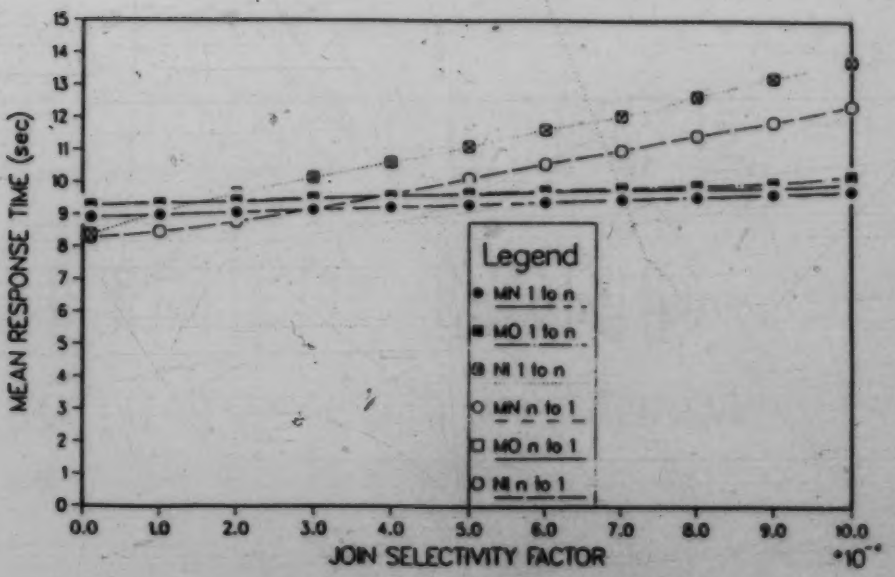

Figure 4.10 Effect of Join Selectivity on Response Time

factor. Letting *many* be located at the inner relation results in the same trends as those of Section 4.1.1. However, if *many* is located at the outer relation, response time for NI is improved. To understand why, recall that tuples that don't match only access the index pages of the inner relation. At low levels of selectivity, the actual number of data pages that are accessed is quite small. As selectivity increases, more data pages are touched and the effects of random probing set in. If *many* is located at the outer relation then *many* calls in a row will access the same tuple and hence the same data page.

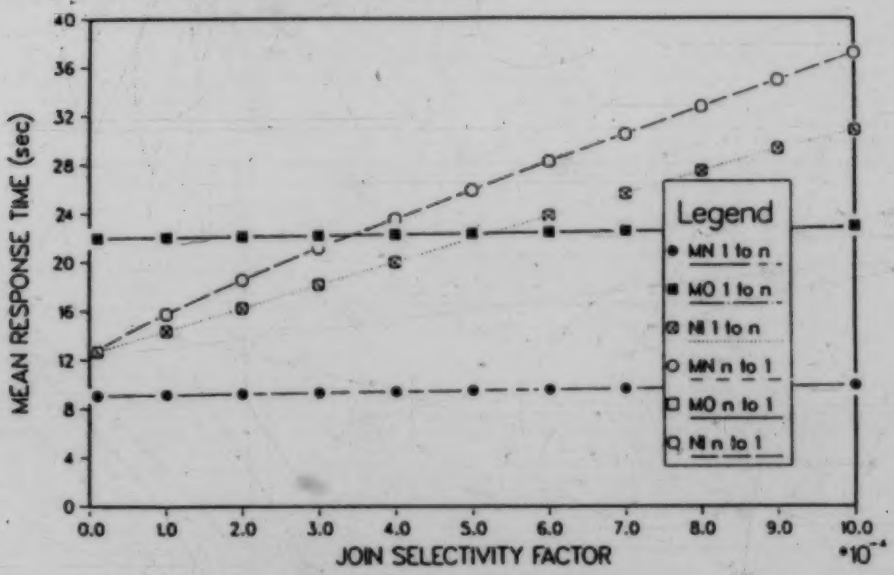

Figure 4.11 Effect of Join Selectivity on Response Time

When buffer space is very limited, as in Figures 4.11 and 4.12, paging behavior becomes very similar to the response time behavior. In addition, MN again becomes the preferred approach.

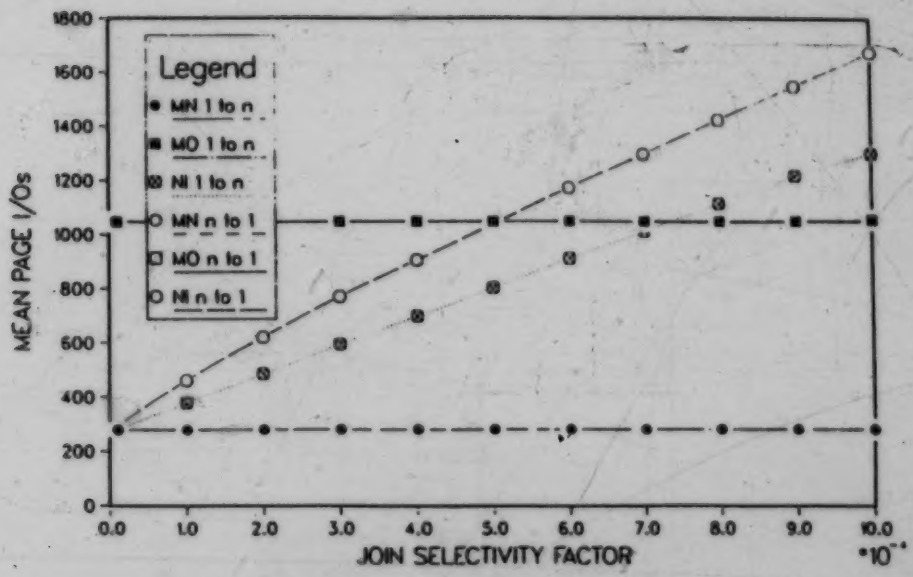

Figure 4.12 Effect of Join Selectivity on Paging

4.1.3. Many-to-Many

Figures 4.13 to 4.15 show the effect of join selectivity on response time, paging, and CPU time, respectively, when buffer size is fixed at 40 pages. Actual values and confidence intervals are provided in Appendix 2.

From Figure 4.13, it is evident that when ample buffer space is available and an index is available on the outer relation, a merge scan making use of the index is preferred (MO). If the inner relation is sorted, no random probing occurs, minimizing the amount of buffer space required. Because of this, ample buffer space is available for the outer relation which minimizes page faults due to random probing and sorting the outer relation becomes more expensive than using the index. If an index is not available then a merge scan that sorts both relations is preferred (MN), again due to the limited use of buffer space by the sorted inner relation during a merge scan.

In Figure 4.16 we have reduced the buffer size to 20 pages, large enough to hold only one of the relations. Now the limited buffer space has caused random probing on the outer relation during the merge scan

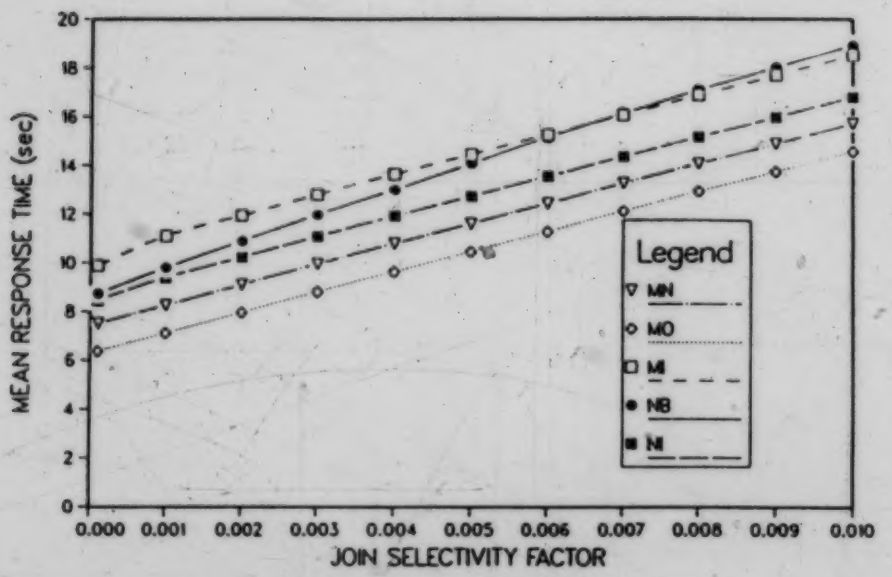

Figure 4.13 Effect of Join Selectivity on Response Time

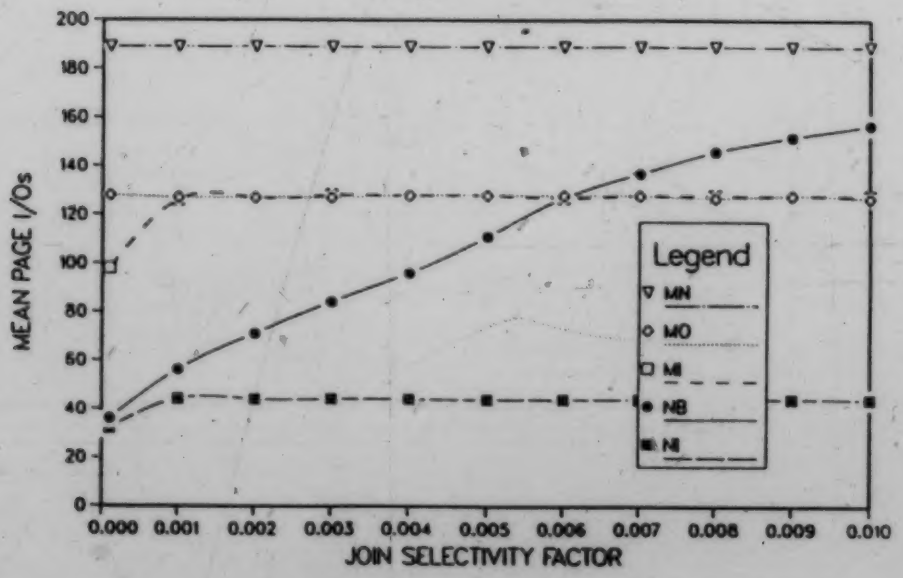

Figure 4.14 Effect of Join Selectivity on Paging

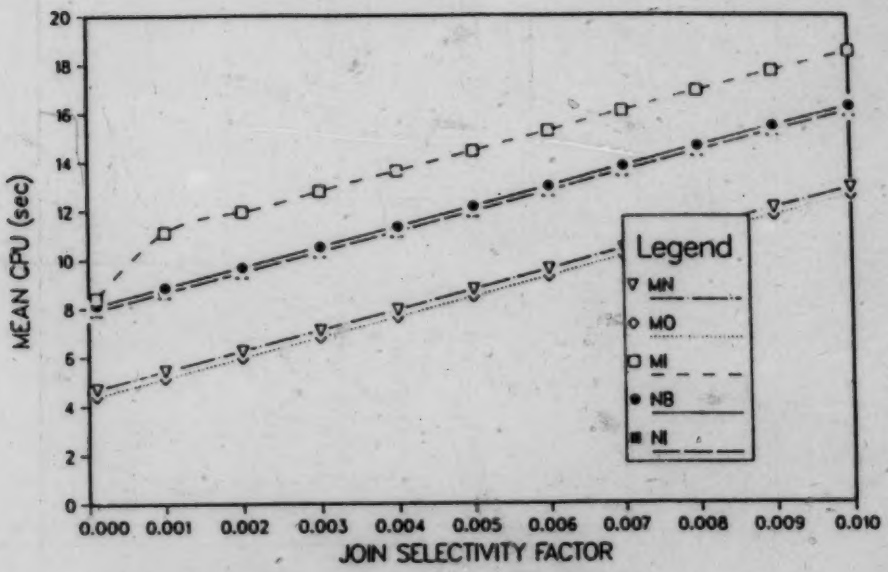

Figure 4.15 Effect of Join Selectivity on CPU Time

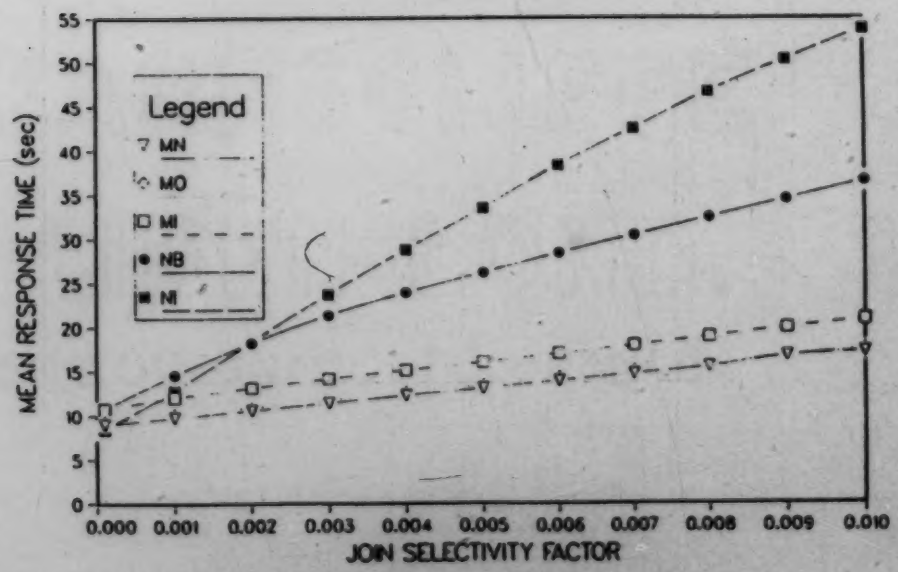

Figure 4.16 Effect of Join Selectivity on Response Time

with an index on the outer and the extra effort to sort the relation become attractive.

At low levels of selectivity, a nested loop with an index on the inner relation is preferred. When selectivity is low, few inner relation tuples are retrieved. If the join attributes don't match, only the index page is accessed while random probing is occurring for the outer relation.

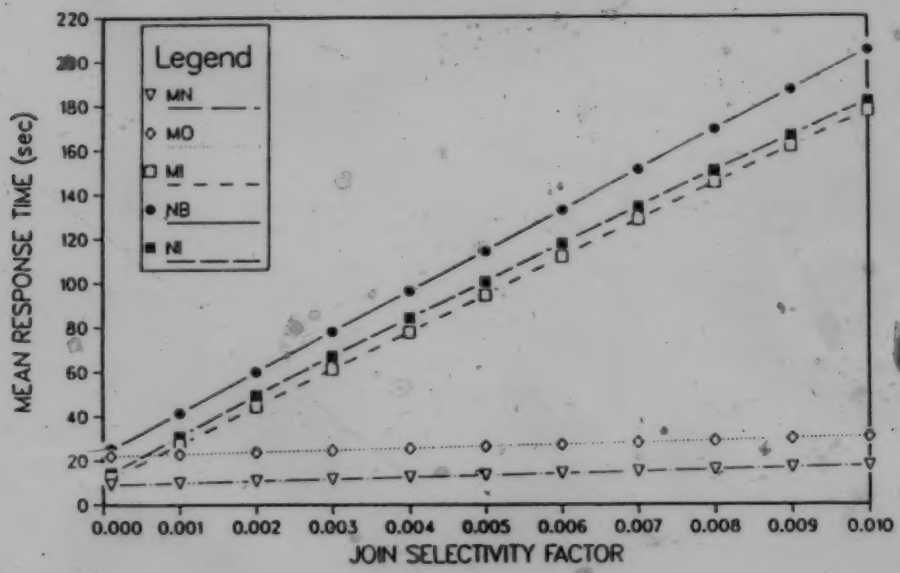

Figure 4.17 Effect of Join Selectivity on Response Time

Figures 4.17 and 4.18 show the case when buffer space is very limited. Now the trends exhibited for paging are very similar to those for response time. The degree of random probing has rendered any approach utilizing an index far too costly. The optimal approach is the merge scan sorting both relations (MN).

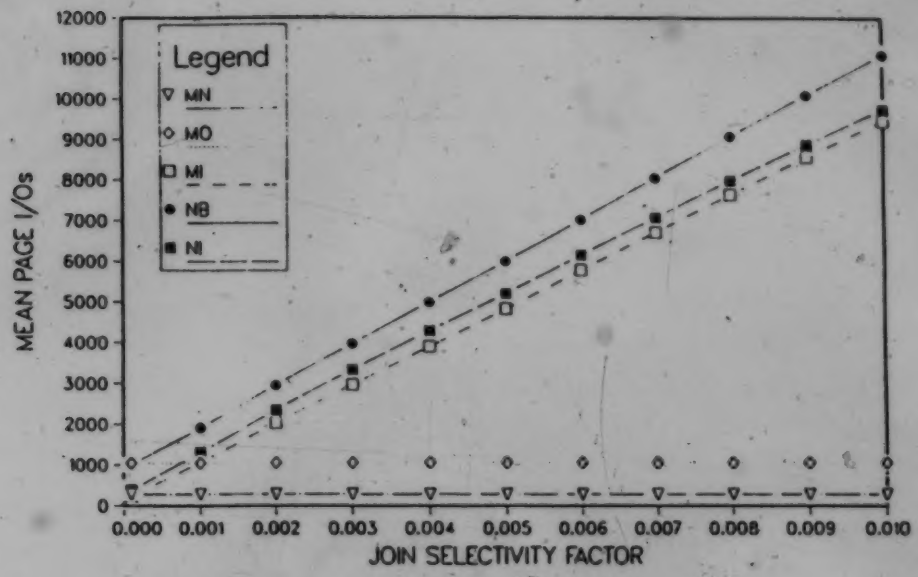

Figure 4.18 Effect of Join Selectivity on Paging

4.2. Joining a Large and a Small Relation

In this section we consider the join between a large and a small relation. As in the previous section, Sections 4.2.1 to 4.2.3 present the results of experiments with attribute relationships of one-to-one, one-to-many, and many-to-many, respectively. For all reported results, the small relation has be set as the outer relation and the large as the inner relation as this produces the lowest response times. The cost of opening scans on the inner relation becomes quite high because a scan is applied for every outer relation tuple. Minimizing the size of the outer relation also minimizes this factor.

4.2.1. One-to-One

In this case the join selectivity factor has an upper bound of 500 / (500 * 1000) or the highest join selectivity is 0.001. Figures 4.19 to 4.21 show the effect of join selectivity on response time, paging, and CPU time, respectively, when buffer size is fixed at 40 pages. Actual values and confidence intervals are provided in Appendix 2.

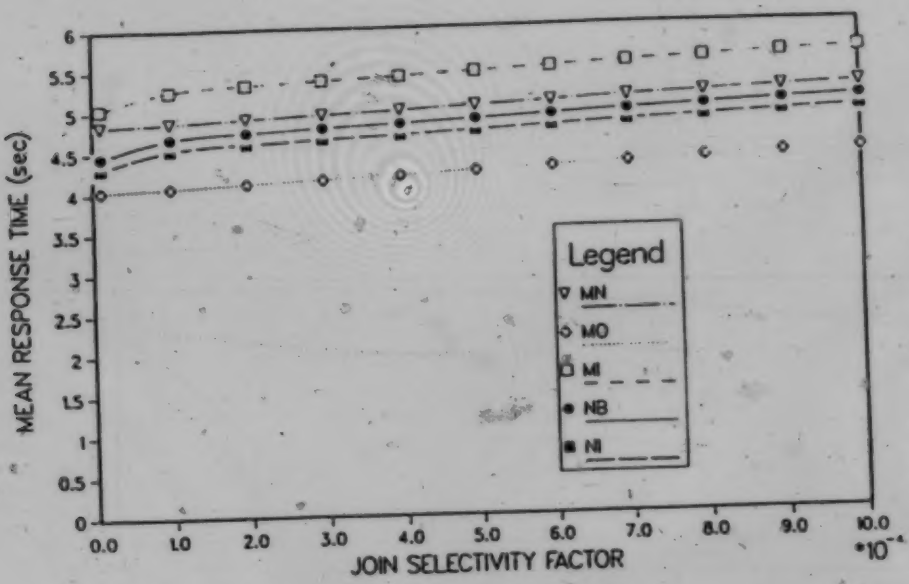

Figure 4.19 Effect of Join Selectivity on Response Time

Again, notice the strong relationship between response time and CPU time. When ample buffer space is available, the level of paging for the two basic scans, index and dbspace, are very similar in that the index scan retrieves each data page once and only once. In light of this, all join methods exhibit similar paging levels during the actual join. However, the merge join based approaches incur additional paging during their sort phase. MI sorts the smaller outer relation, MO sorts the larger inner relation, and MN sorts both. These trends are quite evident in Figure 4.20. The paging of NB is slightly higher than that of NI because of the additional index pages associated with the outer relation.

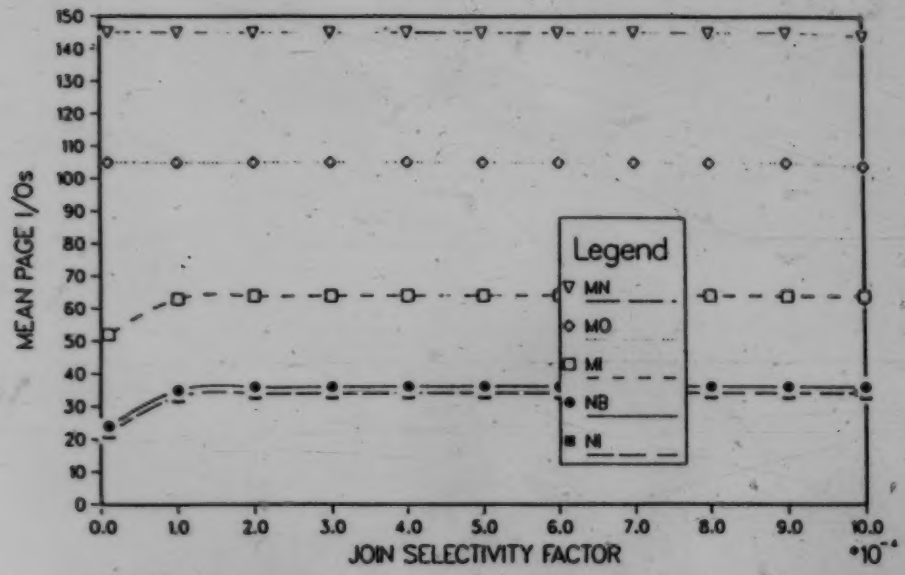

Figure 4.20 Effect of Join Selectivity on Paging

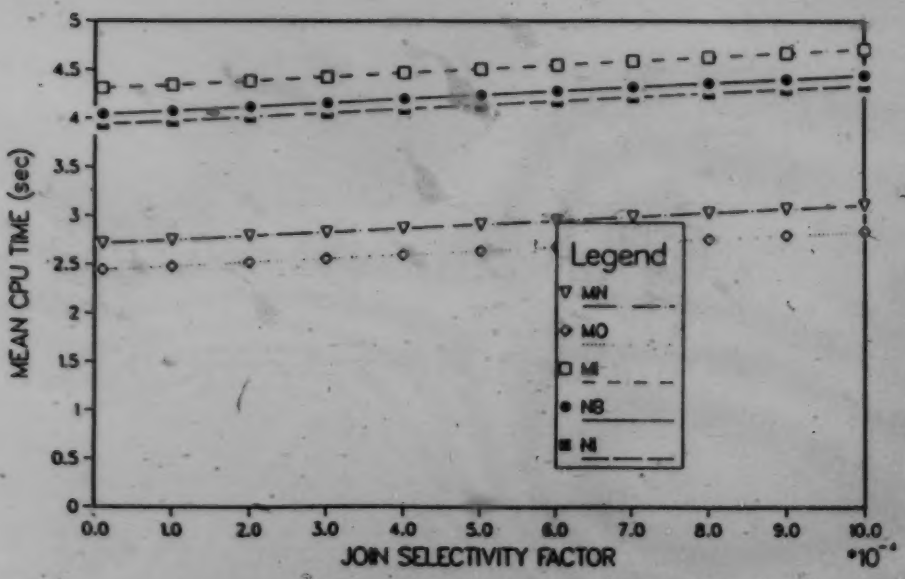

Figure 4.21 Effect of Join Selectivity on CPU Time

Figure 4.21 shows the effect on CPU time. Merge scans with sorted inner relations do not incur the cost of repeatedly opening a scan that is associated with nested loop scan. The cost of maintaining cursors during a merge scan is higher when an index is being applied to the inner relation. The maintenance of a cursor during a dbspace scan is relatively straight forward and inexpensive. As such, MI experiences a much higher CPU time then MN or MI.

Although performing the sort of the inner relation causes more paging than the nested loop approach, the low CPU time for MO has rendered it the optimal approach. If an index is available on the outer relation, a merge scan making use of the index (MO) is preferred.

Although MN experiences a similar advantage in low CPU usage, the paging requirements for sorting the additional relation is high. Although NI incurs higher CPU time, the level of paging is much less. As such, if an index is not available on the outer relation, but is available on the inner relation, then a nested loop join using the inner relation index (NI) is preferred. If no indices are available then the only alternative is a merge scan that sorts both relations (MN).

Figures 4.22 and 4.23 show the effect on response time and paging, respectively, when buffer size has been reduced to 20 pages. Now the buffer is large enough to hold only one of the relations. CPU time has remained the same as in Figure 4.21.

The paging levels for the merge scan based approaches using sorted inner relations change little relatively. However, because of limited buffer space, the cost of sorting the larger inner relation has increased. No change has occurred in the cost of sorting the outer relation. Once the inner relation is sorted, a dbspace scan is used requiring only one buffer space page. This leaves a sufficient amount of buffer space for the entire outer relation.

MI, on the other hand, experiences some effects of random probing. Between a buffer page for the outer sorted relation and the space required for index pages, there is not enough room to hold the entire inner relation. Although NI performs a dbspace scan of the outer relation and makes use of an index on the

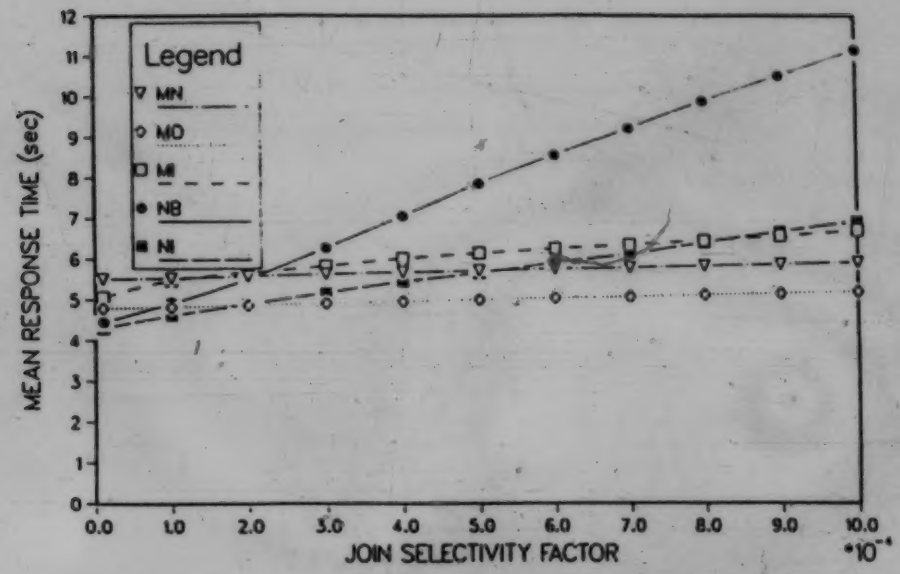

Figure 4.22 Effect of Join Selectivity on Response Time

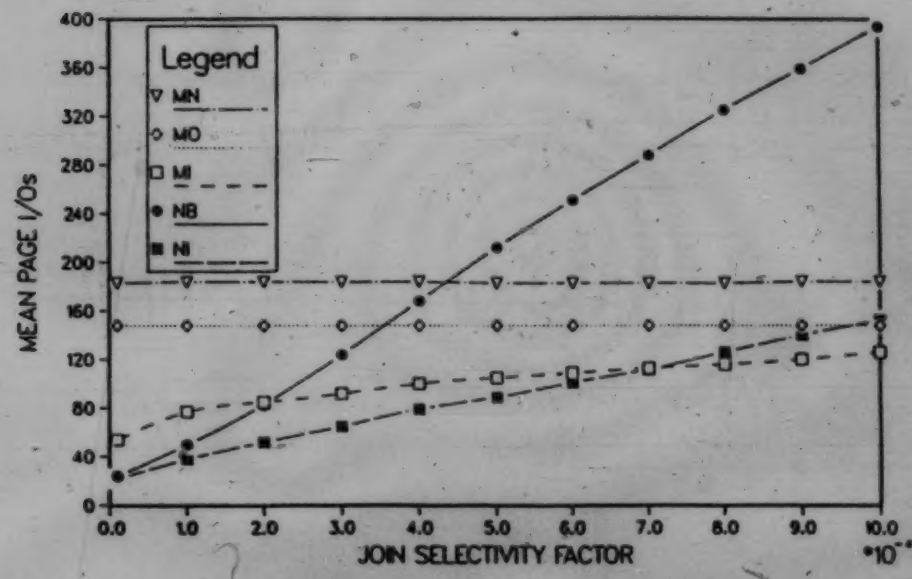

Figure 4.23 Effect of Join Selectivity on Paging

understand why, recall that during MI the outer relation has been sorted on the join attribute, this means that the index values are retrieved in order.

In NI, on the other hand, the outer relation is not sorted and random probing of index pages is responsible for the high increase in page faults. NB makes use of indices on both relations and while selectivity is low, minimal paging is occurring on the inner relation. As selectivity increases, both relations compete for buffer space and page faults increase rapidly.

The high cost of sorting the large inner relation has driven the response times for those approaches uniformly higher leaving nested loop approaches optimal. Unfortunately, as selectivity increases, random probing drives the response times for these approaches higher as well.

As such, at low levels of selectivity a nested loop with an index on the inner relation still offers the best response time. If an index is not available on the inner relation then a merge scan with an index on the outer relation is preferred. If no indices are available, then a merge scan sorting both relations should be used.

If selectivity is not low, then a merge scan with an index on the outer relation is the preferred approach. If no index is available on the outer relation then the merge scan should still be used; however, both relation should be sorted. An exception is if the selectivity is not too high and an index is available on the inner relation. In this case the preferred approach is nested loop as random probing of the inner relation is still at an acceptable level. Recall that other than one page for the outer relation and one page for the index, the rest of the buffer is available for the inner relation.

Figures 4.24 and 4.25 show the case when buffer space is very limited. As expected paging is very similar to response time. The degree of random probing has rendered any approach utilizing an index far too costly except when selectivity is very low. Although the use of an index on the outer relation would cause large amounts of paging, minimal random probing occurs against the inner relation at this level of selectivity. Once the outer relation is sorted, it contributes little to the paging requirements as the scan is reduced to a sequential scan. As such, a merge scan using an index on the outer relation is preferred if the

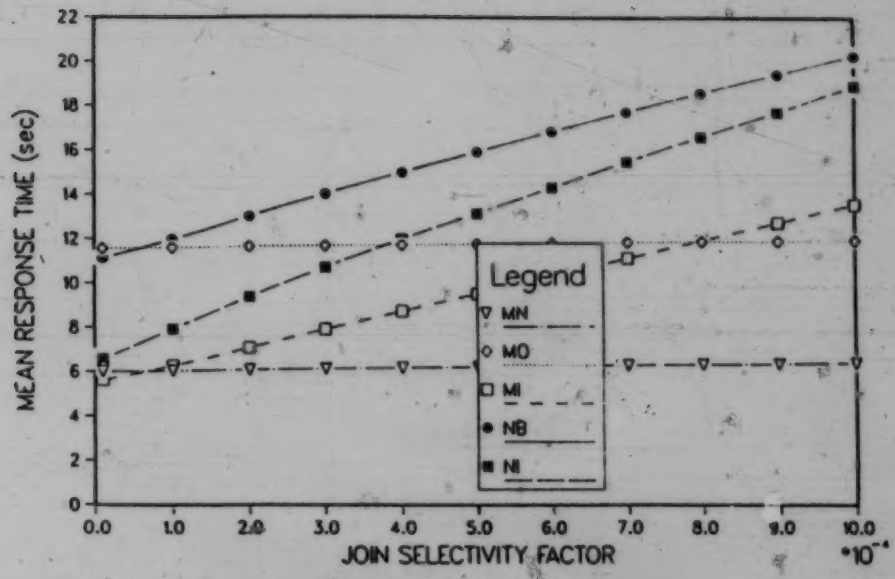

Figure 4.24 Effect of Join Selectivity on Response Time

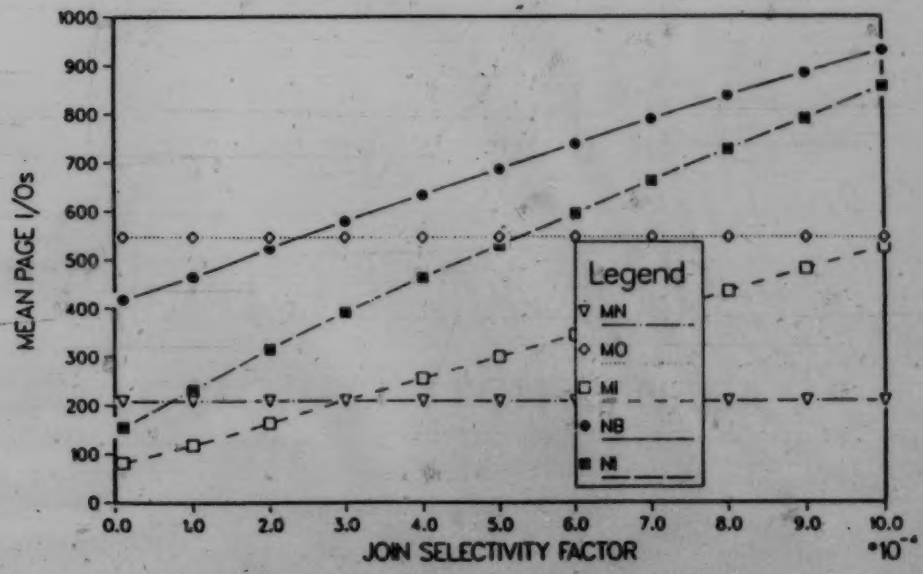

Figure 4.25 Effect of Join Selectivity on Paging

selectivity is very low.

One might expect the nested loop join with an index on the inner relation to offer better response time since the outer relation need not be sorted and the low selectivity is still minimizing the accessing of data pages. However, with such limited buffer space, the accessing of the index pages of the inner relation is causing additional page faults because the tuples of the outer relation are not arriving in index order. With the merge scan, an index page is accessed many times before a new one is required since the outer relation tuples are retrieved in index order.

In all other cases, a merge scan that sorts both relations (MN) offers better response time. Because it makes no use of any indices, it is not subject to the effects of random probing.

4.2.2. One-to-Many

An upper bound of 1000 / (500 * 1000) or a join selectivity of 0.002 now holds. We have restricted this to 0.001 to be consistent with the experiments discussed in the other sections.

Figures 4.26 to 4.28 show the effect of join selectivity on response time, paging, and CPU time, respectively, when buffer size is fixed at 40. Actual values and confidence intervals are provided in Appendix 2.

As in the previous section, with ample buffer space, the location of *many* doesn't matter. The results for both one-to-many and many-to-one offer the same trends when ample buffer space is available. As such, only the case of one-to-many is shown. The availability of ample buffer space makes identification of the optimal approach straightforward. The results are the same as in Section 4.2.1. The fact that *many* tuples are retrieved for a matching outer relation tuple, or visa versa, provides no reduction or change in paging since both relations fit in the available buffer space. If there exists an index on the outer relation, then a merge scan using that index is preferred. If there is no index on the outer relation, but one on the inner relation then a nested loop join (NI) offers the best response time. Finally, if no index is available, then a merge scan that sorts both relations [illegible]

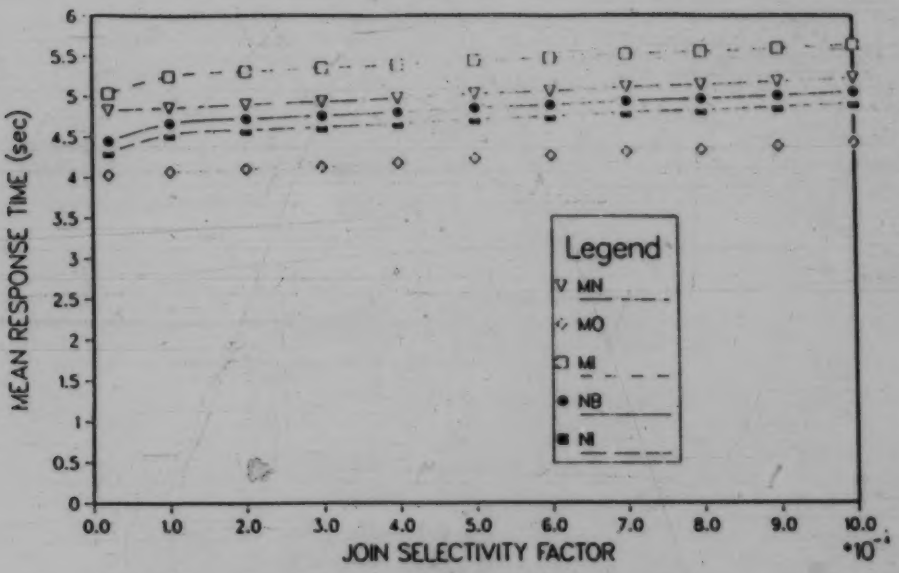

Figure 4.26 Effect of Join Selectivity on Response Time

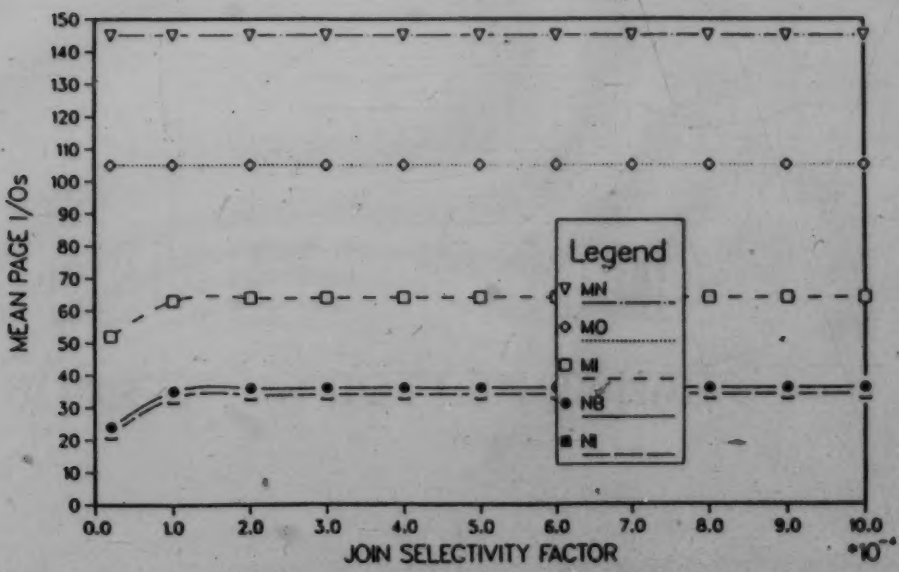

Figure 4.27 Effect of Join Selectivity on Paging

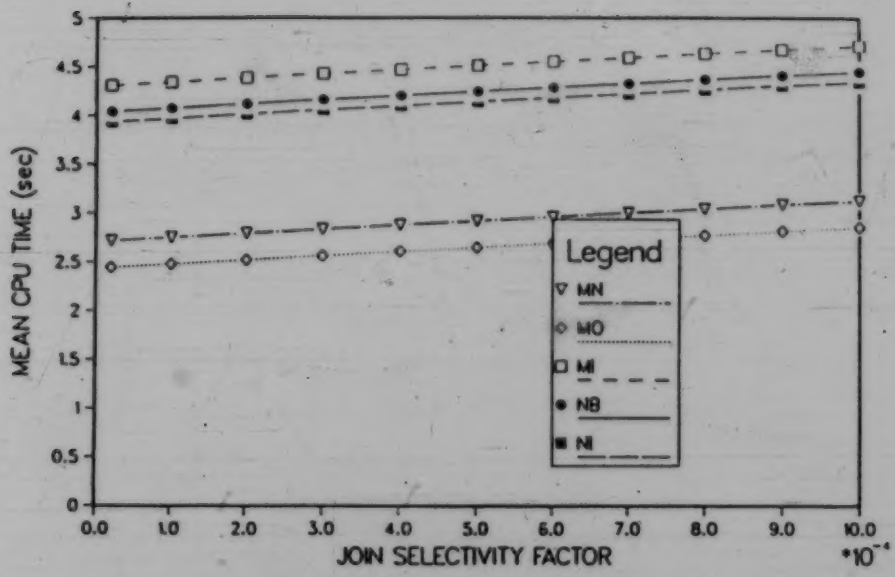

Figure 4.28 Effect of Join Selectivity on CPU Time

When buffer space has been reduced to 20 pages, the location of *many* becomes critical. Now paging can be reduced by retrieving the same tuple *many* times. Figure 4.29 shows the case where *many* is associated with the large inner relation. The result are similar to those of section 4.2.1 at 20 pages. This is because there is still the same amount of buffer space for the inner relation and the same amount of paging is occurring. Since a matching outer relation tuple will match 10 unique inner relation tuples there is the same amount of random probing. However, if *many* is associated with the outer relation, *many* tuples in a row will access the same inner relation tuple.

Figure 4.30 shows the case of many-to-one where *many* is associated with the small outer relation. In this case a merge scan join sorting both relations should only be used if neither relation has an index available as it is the only available option. If selectivity is not high and an index is available on the inner relation then a nested loop using that index is preferred. Previously this was only preferred when selectivity was low. Now, however, the same tuple is retrieved *many* times reducing the level of random prob-

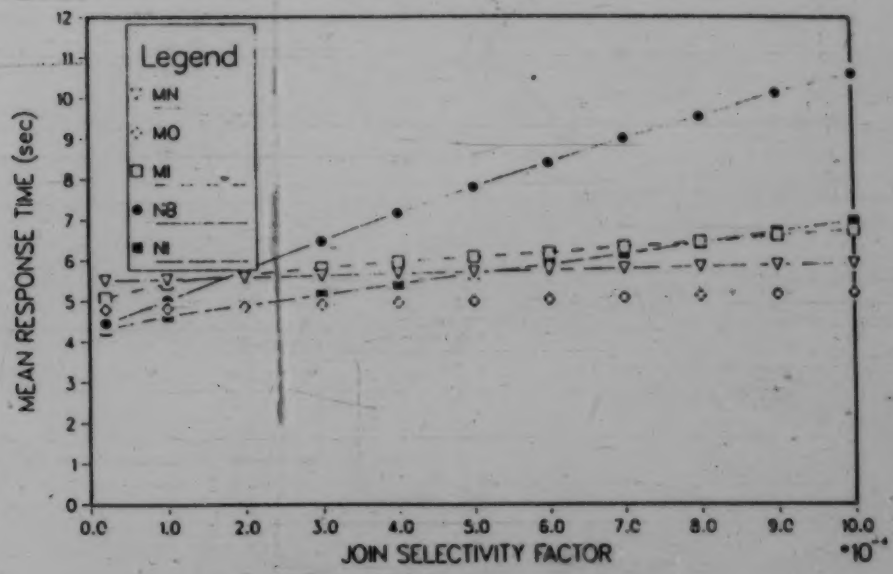

Figure 4.29 Effect of Join Selectivity on Response Time

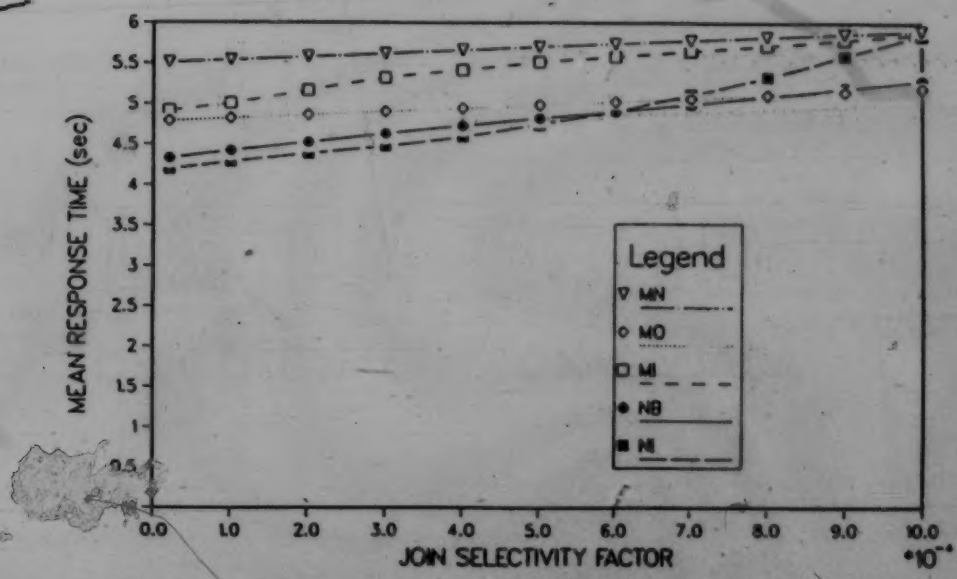

Figure 4.30 Effect of Join Selectivity on Response Time

the trends remain the same. If no index is available on the inner relation or if selectivity is high, and if there is an index on the outer relation, then a merge scan with an index on the outer relation should be used (MO).

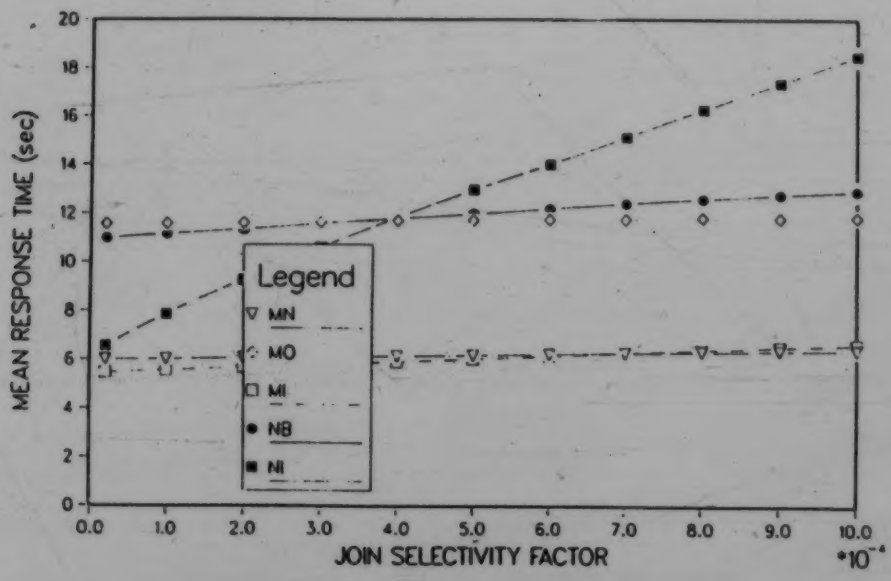

Figure 4.31 Effect of Join Selectivity on Response Time

When buffer space has been limited to four pages, the location of *many* is still critical. When it is associated with the large inner relation, the results are similar to those of Section 4.2.1 with the buffer size limited to 4 pages. Once again this is because the reference patterns against the inner relation shows little change.

Figures 4.31 and 4.32 show the case where *many* is associated with the small outer relation. Buffer space is very limited and any approach that makes use of an index on the outer relation inevitably offers poor response time since random probing causes many page faults. For NI, the random probing of index pages, as well as the random probing of data pages, on the inner relation drives the response time up. Once again, MI is less subject to this phenomenon since the index pages are retrieved in order. As such, if selec-

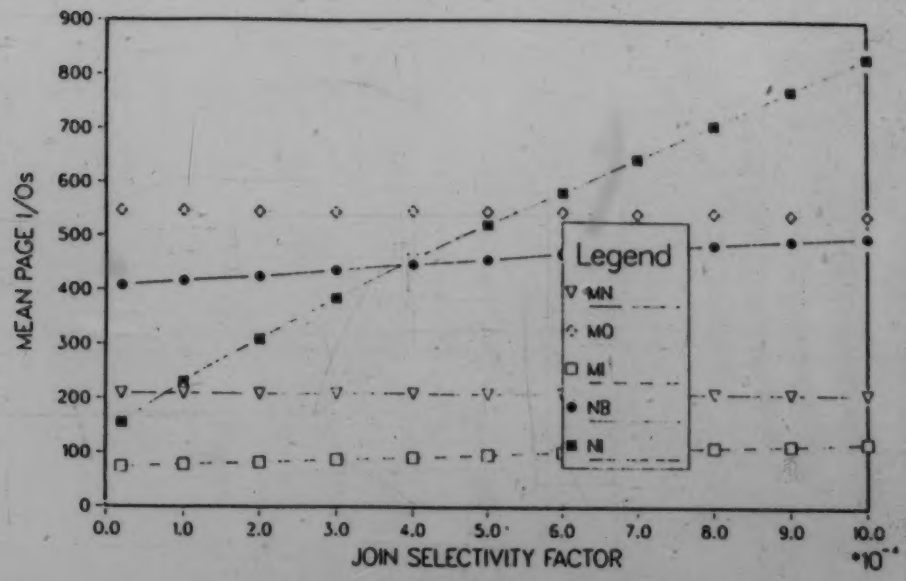

Figure 4.32. Effect of Join Selectivity on Paging

preferred. Otherwise, a merge scan in which both relations are sorted (MN) should be used.

4.2.3. Many-to-Many

In this case the join selectivity factor has an upper bound of (500 * 1000) / (500 * 1000) or the highest join selectivity is 1. Figures 4.33 to 4.35 show the effect of join selectivity on response time, paging, and CPU time, respectively, when buffer size is fixed at 40 pages. Actual values and confidence intervals are provided in Appendix 2.

Once again, the strong relationship between response time and CPU time is evident. From Figure 4.33, when ample buffer space is available, the determination of the best approach is again straightforward. Results are similar to those in Section 4.2.2. The increases are sharper only because the selectivity range is broader. In fact, the conclusions that can be drawn are the same. That is, if an index is available on the outer relation, a merge scan making use of the index is preferred (MO). If an index is not available on the outer relation but is available on the inner then an nested loop join using the inner relation index is pre-

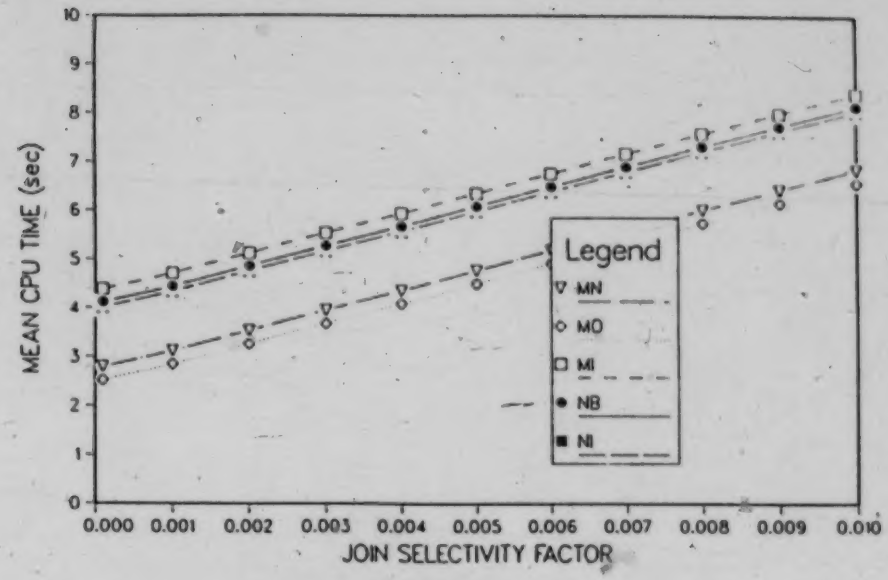

Figure 4.35 Effect of Join Selectivity on CPU Time

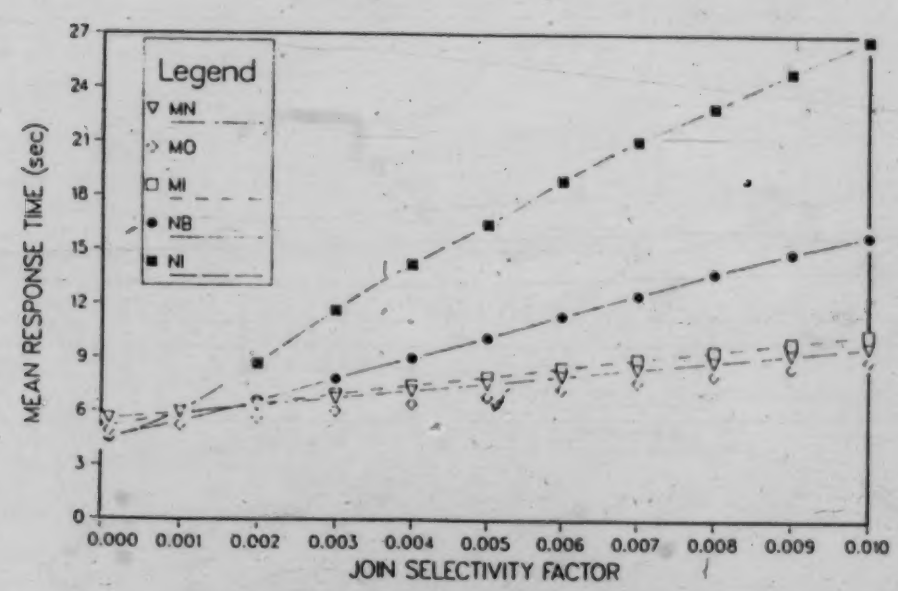

Figure 4.36 Effect of Join Selectivity on Response Time

In Figure 4.36 we have reduced the buffer size to 20 pages, large enough to hold only one of the relations. The trends exhibited are similar to those in Figure 4.33, with a few exceptions. When selectivity is low, few data pages are retrieved minimizing random probing and, of course, page faults when making use of an index on the inner relation. Further reductions can be obtained if outer relation tuples are retrieved in index order. This occurs during a NB, which is the reason why a nested loop join using both indices (NB) when selectivity is very low and an index is available on both relations.

At low levels of selectivity, few inner relation pages are retrieved meaning that few buffer pages are required, leaving more space for the retrieval of outer relation tuples. This additional space makes applying the index on the outer more attractive then sorting. As selectivity increases, so too does random probing against the inner relation leaving less room for the outer. Now sorting the outer relation (MI) becomes more attractive because of the page faults incurred during an index scan of the outer relation. The merge scan based approaches exhibit the same trends as in Figure 4.33. That is, if an index is only available on the outer relation then a merge scan with an index on the outer relation (MO) should be used. Otherwise a merge scan that sorts both relation is preferred.

Figures 4.37 and 4.38 show the case when buffer space is very limited. As expected, paging is very similar to response time. Once again, a merge scan that sorts both relations (MN) is always preferred.

4.3. Summary

We divide the summary in three parts, each based on the availability of buffer space. It should be kept in mind that, for joins, the smaller of the two relations is assigned as the outer relation. When ample buffer space is available the following conclusions are drawn:

```
If an index is available on the outer relation then
    use MO
Else If both relations are the same size then
    use MN
Else If the relations are different sizes then
    If an index is available on the inner relation then
        use NI
```

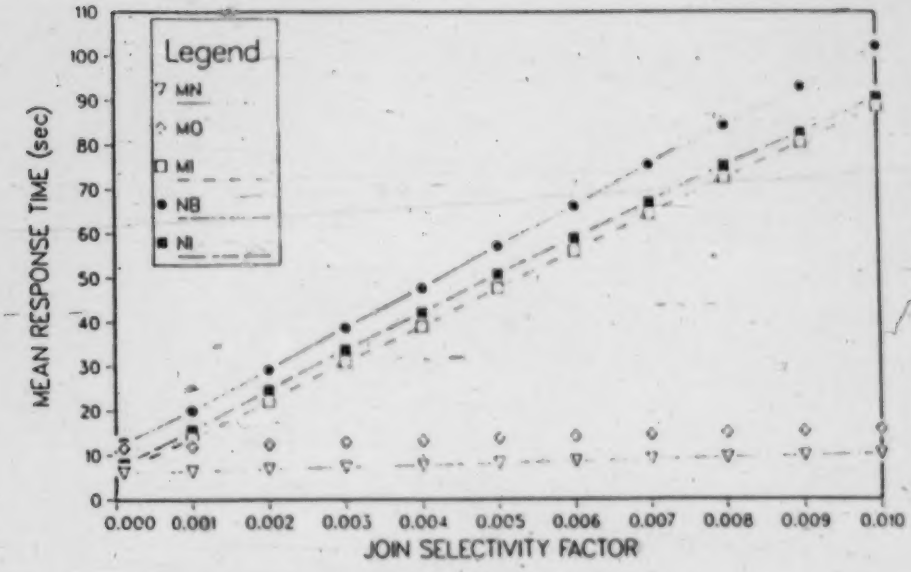

Figure 4.37 Effect of Join Selectivity on Response Time

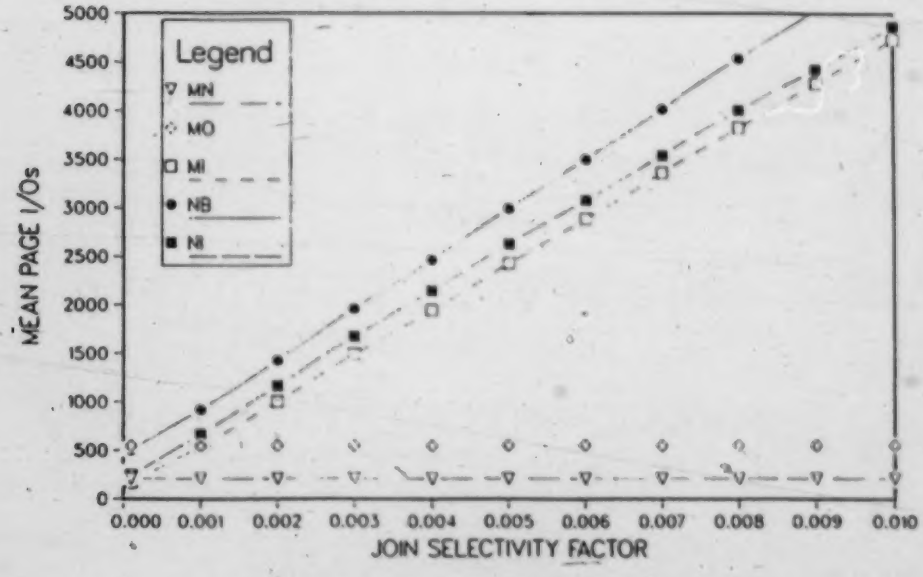

Figure 4.38 Effect of Join Selectivity on Paging

```
    Else
        use MN
    Endif
Endif
```

When there is enough buffer space for the inner relation the following conclusions are drawn:

```
If both relations are the same size then
    If selectivity is low then
        If an index is available on the inner relation then
            If the join attribute relationship is not even (i.e. not one-to-one or many-to-many) then
                Let n be associated with the outer relation
            Endif
            use NI
        Else
            use MN
        Endif
    Else
        use MN
    Endif
Else If the relations are of different sizes then
    If the join attribute relationship is 1 to 1 then
        If the selectivity is low and an index is available on the inner relation then
            use NI
        Else If an index is available on the outer relation then
            use MO
        Else
            use MN
        Endif
    Else If the join attribute relationship is 1-to-many then
        If n is associated with the larger inner relation then
            If the selectivity is low and an index is available on the inner relation then
                use NI
            Else If an index is available on the outer relation then
                use MO
            Else
                use MN
            Endif
        Else If many is associated with the smaller outer relation then
            If the selectivity is not high then
                If an index is available on the inner relation then
                    use NI
                Else If an index is available on the outer relation then
                    use MO
                Else
                    use MN
                Endif
            Else If the selectivity is high then
                If an index is available on the outer relation then
```

```
                    use MO
                Else If an index is available on the inner relation then
                    use NI
                Else
                    use MN
                Endif
            Endif
        Endif
    Else If the join attribute relationship is many-to-many then
        If the selectivity is low then
            If an index is available on both relations then
                use NB
            Else If an index is available on the inner relation then
                use NI
            Else If an index is available on the outer relation then
                use MO
            Else
                use MN
            Endif
        Else If the selectivity is not low then
            If an index is available on the outer relation then
                use MO
            Else
                use MN
            Endif
        Endif
    Endif
Endif
```

When buffer space is limited such that neither relation with fit completely in the buffer the following conclusions are drawn:

```
If both the relations are the same size then
    use MN
Else If the relations are of different sizes then
    If the join attribute relationship is 1 to 1 then
        If the selectivity is low then
            If an index is available on the inner relation then
                use MI
            Else
                use MN
            Endif
        Else
            use MN
        Endif
    Else If the join attribute relationship is one-to-many then
        If many is associated with the larger inner relation then
            If the selectivity is low then
```

```
            If an index is available on the inner relation then
                use MI
            Else
                use MN
            Endif
        Else
            use MN
        Endif
    Else If many is associated with the smaller outer relation then
        If the selectivity is not high then
            If an index is available on the inner relation then
                use MI
            Else
                use MN
            Endif
        Else
            use MN
        Endif
    Endif
Else If the join attribute relationship is many-to-many then
    use MN
Endif
Endif
```

Chapter 5

CONCLUSIONS

It appears that heuristic query optimizers can provide a valid alternative to other approaches, such as priority orderings and cost functions. Simple priority orderings do not provide the flexibility for considering alternative approaches to service a query. We have found no universal priority ordering for the specific database environment that is considered in this study. Each investigated approach offers superior performance in different situations. As such, no ordering exists that will determine an optimal approach under all circumstances.

Elaborate cost functions, on the other hand, are very complex. Even though no information is available as to the overhead of evaluating these cost functions, the assumption that this cost is amortized over many runs indicates that it is significant. For example, recall that the cost function that R* uses for local queries in given as

$$Wcpu * (\#_instrs) + Wi/o * (\#_i/o).$$

However, the number of instructions ($\#_instrs$) for a simple scan, which is only one of the terms, is given in [MaL86b] as;

$$CPU_{scan} = OPENINST + FF_{RDOM} * TABLECARD * (TUPLINST + SARGINST) +$$

$$FF_{SARG} * TABLECARD * (RSSCALL + RESIDINST)$$

where

$$TUPLINST = \begin{cases} 0 & \text{if } no\ index \\ BUFFERINST & \text{if } only\ index\ pages\ are\ accessed \\ 2*BUFFERINST & otherwise \end{cases}$$

$$RESIDINST = \begin{cases} 0 & \text{if } \#RES = 0 \\ RESID_SETUP + \Sigma ARINST_i & \text{if } \#RES > 0 \end{cases}$$

Furthermore, the terms in these equations have to be estimated, which contributes to the complexity. The terms in the above equation are the following:

OPENINST - the number of instructions to OPEN and CLOSE the scan;

SARGINST - the number of instructions to apply a boolean (or SARGable) predicate (a linear function of the number of SARGable predicates which is a query dependent variable);

RSSCALL - the number of instructions to invoke the RSS (storage subsystem);

RESID_SETUP - the number of instructions overhead for applying residual predicates

$ARINST_i$ - the number of instructions for calculating arithmetic expressions in residual predicate i (a linear function of the number of such expressions in predicate i which is a query dependent variable);

TABLECARD - the cardinality of the relation;

FF_{KDOM} - the estimated filter factor of any predicate that can be applied using the index, thus limiting the index scan to a subtree; = 1 otherwise or if this is a Dbspace scan;

FF_{SARG} - the estimated filter factor of all SARGable predicates;

#RES - the number of residual predicates.

Consider a query which involves a simple selection and assume that ample buffer space is available. The same complicated cost function needs to be evaluated for the optimization of this query.. The heuristic, however, is as follows:

```
If no index is available on the selection predicate then
        use a Dbspace scan
Else if (the number of data pages touched during an index scan) + (the number of index pages) <
        (the number of data pages occupying tuples of the relation) then
        use an index scan
Else
        use a Dbspace scan
Endif
```

Note that the cost function presented here is only for CPU instructions for simple queries. We have not considered the cost functions for page fetches, sorts, and joins. The cost functions that are commonly used do not consider the effect of buffer size. Given an operating environment, a system configuration, and the set of query processing algorithms and options available, it is possible to derive a set of heuristic rules to govern the selection process.

For the scenario presented in this thesis, the heuristic rules themselves involve the consideration of relatively few conditions. For more complicated cases, the complexity is a function of the number of conditions required to uniquely determine the optimal approach.

A variety of conclusions can be drawn from the results of this study. It is quite evident that estimation of buffer availability is just as critical to query optimization as is the selectivity factor. The sorting of relations for merge scans represents a dominant cost factor in query processing. Our investigation shows that the optimal approach, for those considered cases, occur under clearly defined conditions and that once those conditions are identified, they offer a simple determining criterion.

The results presented have several applications. First, they can be applied directly to form the basis of a query optimizer. The fact that buffer contention is included in the rules should provide a vital factor that is missing in the cost equations (see [MaL86b] for a discussion). Second, they can be used to prune the search space of alternative strategies that need to be analyzed by a conventional query optimizer. Finally, they can be applied in semantic query optimization algorithms, similar to that presented by King [Kin84], to further restrict queries generated.

Since this thesis constitutes an initial investigation into this methodology, there are many ways in which the work presented here may be expanded. The study should be extended to investigate the effect of system load, for example. Although varying buffer size can represent the effect of shared buffer contention, the competition for CPU will, inevitably, have a profound effect on CPU intensive query processing strategies. For example, nested loop joins pay an enormous CPU cost while re-opening the new scans of the inner relation. The investigation should be extended to multiple joins, other system configurations, and possibly to distributed systems.

Although the inclusion of more recent query processing algorithms is most appealing, their placement in and integration into the database management system would be procrustean. It is necessary to have an existing system to start with as we are not really at liberty to modify the configuration and/or algorithm.

In addition, algorithms that work well with internal data structures running by themselves as a Pascal program are often not easily implemented and integrated into a DBMS. The mechanics and peculiarities of a specific system architecture can often be a restriction. The system we have adopted in this study has many quirks that are probably there because of implementation restrictions.

For example, the fact that operations are done tuple-at-a-time on base relations does not permit nested loop joins to operate with pages or blocks. Semi-joins should eliminate the high degree of buffer contention encountered during an nested loop join making use of an index on both relations. The contention is associated with random probing on the outer relation's data pages. Ideally, only the indices of the outer relation are needed to retrieve the inner relation tuples. A semi-join would do this and then apply the resulting relation back against the outer. Unfortunately, this requires multiple scans, involving temporary relations which, apparently, is not provided for in R*. Another alternative is to only retrieve the index value from the outer relation and if there is a match on the inner relation then, and only then, retrieve the outer relation tuple. This, unfortunately disrupts the scan of the outer relation and is, again, not provided for in R*.

In addition, it appears that scans against outer and inner relations are not done independently and, as such, a page request against the inner relation forces operations against the outer relation to wait as well. Again, this is likely a restriction of the architecture of R*.

The work reported here points to other weaknesses and the importance of research in these areas is evident. One such area is in the development of alternative sorting algorithms that reduce paging directly or, more importantly, by minimizing the potential of page faults. The importance of sorting relations for joins is evident. Lower response times for those approaches involving sorting will greatly simplify the hueristic rules by eliminating those cases where other approaches must be considered. In addition, buffer management policies, due to their profound effects, warrant further investigation. Sacco and Schkolnick [SaS86] have stressed the need for the revision of traditional cost measures and buffer awareness with respect to access strategies. If DBMSs based on non-procedural languages are going to achieve

the high performance that is demanded of them, it is these areas that require a lot more attention then they have received in the past

References

[AHU74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[ABC76] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade and V. Watson, System R: Relational Approach to Database Management, *ACM Trans. Database Systems 1*, 2 (June 1976), 97-137.

[ASK80] M.M. Astrahan, M. Schkolnick and W. Kim, Performance of the System R Access Path Selection Mechanism, *Information Processing 80*, 1980, 487-491.

[Bat85] D.S. Batory, Modeling the Storage Architectures of Commerical Database Systems, *ACM Trans. Database Systems 10*, 4 (Dec. 1985), 463-528.

[BaM72] R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indexes, *Acta Informatica 1*, (1972), 173-189.

[BeS86] S. Bergamaschi and M.R. Scalas, Choice of the Optimal Number of Blocks for Data Access by an Index, *Information Systems 11*, 3 (1986), 199-209.

[BGW81] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve and J.B. Rothnie, Query Processing in a System for Distributed Databases, *ACM Trans. Database Systems 6*, 4 (Dec. 1981), 602-625.

[BBD83] D. Bitton, H. Boral, D.J. DeWitt and W.K. Wilkinson, Parell Algorithms for the Execution of Relational Database Operations, *ACM Trans. Database Systems 8*, 3 (Sep. 1983), 324-353.

[BlE77] M.W. Blasgen and K.P. Eswaran, Storage and Access in Relational Data Bases, *IBM SYST. J.*, 1977, 363-377.

[ChE86] T. Chao and C.J. Egyhazy, Estimating Temporary File Sizes in Distributed Relational Database Systems, *Proc. of the Int. Conf. on Data Engineering*, 1986, 4-12.

[ChD86] H. Chou and D.J. DeWitt, An Evaluation of Buffer Management Strategies of Relational Database Systems, *Algorithmica*, 1986, 311-336.

[DSH82] D. Daniels, P. Selinger, L. Haas, B. Lindsay, C. Mohan, A. Walker and P. Wilms, An Introduction to Distributed Query Compilation in R* , in *Distributed Data Bases*, North-Holland, 1982, 291-309.

[Dan82] D. Daniels, Query Compilation in a Distributed Database System, Tech. Report Number RJ3423, IBM San Jose Research Lab, Mar. 1982.

[Dat77] C.J. Date, *An Introduction to Database Systems*, Addison Wesley, 1977.

[Dat87] C.J. Date, *A Guide to the SQL Standard*, Addison Wesley, 1987.

[DeW79] D.J. DeWitt, Query Execution in DIRECT, *Proc. of ACM-SIGMOD Int. Conf. on Management of Data* , 1979, 13-22.

[EfL84] W. Effelsberg and M.E.S. Loomis, Logical, Internal, and Physical Reference Behavior in CODASYL Database Systems, *ACM Trans. Database Systems 9*, 2 (June 1984), 187-213.

[EfH84] W. Effelsberg and T. Haerder, Principles of Database Buffer Management, *ACM Trans. Database Systems 9*, 4 (Dec. 1984), 560-595.

[FLW78] E.B. Fernandez, T. Lang and C. Wood, Effect of Replacement Algorithms on a Paged Buffer Database System, *IBM J. Res. Develop 22*, 2 (Mar. 1978), 185-196.

[HaF86] R.B. Hagmann and D. Ferrari, Perfromance Analysis of Several Back-End Database Architectures, *ACM Trans. Database Systems 11*, 1 (Mar. 1986), 1-26.

[HaS79] P. Hawthorn and M. Stonebraker, Performance Analysis of a Realtional Database Management System , *Proc. of ACM-SIGMOD Int. Conf. on Management of Data* , 1979, 1-12.

[Kin84] J.J. King, *Query Optimization by Semantic Reasoning*, UNI Research Press, Computer Science: Artificial Intelligence, 1984.

[LWF77] T. Lang, C. Wood and I.B. Fernandez, Database Buffer Paging in Virtual Storage Systems, *ACM Trans. Database Systems* 2, 4 (Dec. 1977), 339-351.

[LeC86] T.J. Lehman and M.J. Carey , Query Processing in Main Memory Database Management Systems, *Proc. of ACM-SIGMOD Int. Conf. on Management of Data* , 1986.

[Loh87] G.M. Lohman, Personal Correspondence, , Jan. 1987.

[MaL86a] L.F. Mackert and G.M. Lohman, R* Optimizer Validation and Perfromance Evaluation for Distributed Queries, *Proc. of Int. Conf. on Very Large Data Bases*, 1986, 149-159.

[MaL86b] L.F. Mackert and G.M. Lohman, R* Optimizer Validation and Perfromance Evaluation for Local Queries, *Proc. of ACM-SIGMOD Int. Conf. on Management of Data* , 1986, 84-95.

[Mar83] S.T. March, Techniques for Structuring Database Records, *Computing Surveys 15*, 1 (Mar. 1983), 45-79.

[Ng82] P. Ng, Distributed Compilation and Recompilation of Database Queries, Tech. Report Number RJ3375, IBM San Jose Research Lab, Jan. 1982.

[OzV88] T. Ozsu and P. Valduriez, *Principals of Distributed Systems*, Prentice Hall (forth coming), 1988.

[PiC84] G. Piatetsky-Shapiro and C. Connell, Accurate Estimation of the Number of Tuples Satisfying a Condition, *Proc. of ACM-SIGMOD Int. Conf. on Management of Data* , 1984, 256-276.

[SaS86] G.M. Sacco and M. Schkolnick, Buffer Management in Relational Database Systems, *ACM Trans. Database Systems 11*, 4 (Dec. 1986), 473-498.

[Sac86] G.M. Sacco, Fragmentation: A Technique for Efficient Query Processing, *ACM Trans. Database Systems 11*, 2 (June 1986), 113-133.

[SAC79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie and T.G. Price, Access Path Selection in a Relational Database Management System, *Proc. of ACM-SIGMOD Int. Conf. on Management of Data* , 1979, 23-34.

[SeA80] P.G. Selinger and M. Adiba, Access Path Selection in Distributed Database Management System, Tech. Report Number RJ2883, IBM San Jose Research Lab, Aug. 1980.

[SeC77] D.G. Severance and J.V. Carlis, A Practical Approach to Selecting Record Access Paths, *Computing Surveys* 9, 4 (Dec. 1977), 259-272.

[Sha86] L.D. Shapiro, Join Processing in Database Systems with Large Main Memories, *ACM Trans. Database Systems 11*, 3 (Sep. 1986), 239-264.

[SWK76] M. Stonebraker, E. Wong, P. Kreps and G. Held, The Design and Implementation of INGRES, *ACM Trans. Database Systems 1*, 3 (Sep. 1976), 189-222.

[Sto81] M. Stonebraker, Operating System Support for Database Management, *Comm. ACM 24*, 7 (July 1981), 412-418.

[Ull82] J.D. Ullman, *Principles of Database Systems*, Computer Science Press, 1982.

[Val87] P. Valduriez, Join Indices, *ACM Trans. Database Systems 12*, 2 (Dec. 1987), 218-246.

[Yao77a] S.B. Yao, *Approximating Block Accesses in Database Orginizations*, Comm. ACM, Apr. 1977.

[Yao77b] S.B. Yao, An Attribute Based Model for Database Access Cost Analysis, *ACM Trans. Database Systems* 2, 1 (Mar. 1977), 45-67.

Appendix A1

Simple Query Results

The performance results obtained for the simple queries are listed in this appendix.

1.1. Type 1 Queries

Mean Response Time				
SELECTIVITY (%)	1000		5000	
	DBSPACE	INDEX	DBSPACE	INDEX
1	0.388	0.189	1.864	0.790
10	0.444	0.462	2.141	5.893
20	0.506	0.545	2.450	11.570
30	0.567	0.644	2.758	17.102
40	0.629	0.726	3.067	22.728
50	0.691	0.808	3.375	28.361
60	0.752	0.908	3.684	34.053
70	0.814	0.990	3.992	39.695
80	0.876	1.089	4.301	45.403
90	0.937	1.171	4.609	51.117
100	0.999	1.253	4.918	56.876

Table A1.1 Response Time vs Selectivity

Confidence Intervals (95%)				
SELECTIVITY (%)	1000		5000	
	DBSPACE	INDEX	DBSPACE	INDEX
1	±0	±0.009	±0	±0.020
10	±0	±0.002	±0	±0.101
20	±0	±0	±0	±0.119
30	±0	±0	±0	±0.135
40	±0	±0	±0	±0.159
50	±0	±0	±0	±0.155
60	±0	±0	±0	±0.144
70	±0	±0	±0	±0.191
80	±0	±0	±0	±0.235
90	±0	±0	±0	±0.249
100	±0	±0	±0	±0.251

Table A1.2 Confidence Interval for Response Time vs Selectivity

Mean Page I/O				
SELECTIVITY (%)	1000		5000	
	DBSPACE	INDEX	DBSPACE	INDEX
1	20	9	100	43
10	20	21	100	321
20	20	21	100	631
30	20	22	100	932
40	20	22	100	1239
50	20	22	100	1546
60	20	23	100	1857
70	20	23	100	2165
80	20	24	100	2476
90	20	24	100	2788
100	20	24	100	3103

Table A1.3 Page Requests vs Selectivity

Mean CPU Time				
SELECTIVITY (%)	1000		5000	
	DBSPACE	INDEX	DBSPACE	INDEX
1	0.048	0.032	0.164	0.065
10	0.104	0.106	0.441	0.434
20	0.166	0.188	0.750	0.845
30	0.227	0.270	1.058	1.255
40	0.289	0.352	1.367	1.666
50	0.351	0.434	1.675	2.076
60	0.412	0.517	1.984	2.487
70	0.474	0.599	2.292	2.897
80	0.536	0.681	2.601	3.308
90	0.597	0.763	2.909	3.718
100	0.659	0.845	3.218	4.129

Table A1.4 CPU vs Selectivity

1.2. Type 2 Queries

Mean Response Time			
SELECTIVITY (%)	1000		
	INDEX	AFTER SORT	BEFORE SORT
1	0.189	1.767	0.535
10	0.462	1.824	0.743
20	0.545	1.888	0.965
30	0.644	1.952	1.182
40	0.726	2.016	1.398
50	0.808	2.080	1.703
60	0.908	2.144	1.910
70	0.990	2.207	2.113
80	1.089	2.272	2.317
90	1.171	2.336	2.979
100	1.253	2.400	3.480

Table A1.5 Response Time vs Selectivity (1000)

Mean Response Time			
SELECTIVITY (%)	5000		
	INDEX	AFTER SORT	BEFORE SORT
1	0.790	14.981	2.037
10	5.893	15.421	3.187
20	11.570	15.911	4.886
30	17.102	16.401	7.376
40	22.728	16.891	9.456
50	28.361	17.381	11.46
60	34.053	17.870	15.534
70	39.695	18.360	17.737
80	45.403	18.850	19.900
90	51.117	19.340	22.135
100	56.876	19.813	25.270

Table A1.6 Response Time vs Selectivity (5000)

Confidence Intervals (95%)			
SELECTIVITY (%)	1000		
	INDEX	AFTER SORT	BEFORE SORT
1	±0.009	±0.004	±0
10	±0.002	±0.004	±0.001
20	±0	±0.004	±0.001
30	±0	±0.004	±0.002
40	±0	±0.004	±0.002
50	±0	±0.004	±0.007
60	±0	±0.004	±0.004
70	±0	±0.004	±0.005
80	±0	±0.004	±0.004
90	±0	±0.004	±0.006
100	±0	±0.004	±0.004

Table A1.7 Confidence Interval for Response Time vs Selectivity (1000)

Confidence Intervals (95%)			
SELECTIVITY (%)	5000		
	INDEX	AFTER SORT	BEFORE SORT
1	±0.020	±0.004	±0.004
10	±0.101	±0.004	±0.001
20	±0.119	±0.004	±0.005
30	±0.135	±0.004	±0.008
40	±0.159	±0.004	±0.011
50	±0.155	±0.004	±0.015
60	±0.144	±0.004	±0.012
70	±0.191	±0.004	±0.015
80	±0.235	±0.004	±0.012
90	±0.249	±0.004	±0.012
100	±0.251	±0.004	±0.004

Table A1.8 Confidence Interval for Response Time vs Selectivity (5000)

Mean Page I/O			
SELECTIVITY (%)	1000		
	INDEX	AFTER SORT	BEFORE SORT
1	9	84	24
10	21	84	28
20	21	84	36
30	22	84	44
40	22	84	52
50	22	84	70
60	23	84	80
70	23	84	90
80	24	84	100
90	24	84	112
100	24	84	124

Table A1.9 Page Requests vs Selectivity (1000)

Mean Page I/O			
SELECTIVITY (%)	5000		
	INDEX	AFTER SORT	BEFORE SORT
1	43	802	104
10	321	811	150
20	631	821	205
30	932	831	318
40	1239	841	409
50	1546	851	495
60	1857	861	696
70	2165	871	796
80	2476	881	898
90	2788	891	1000
100	3103	900	1100

Table A1.10 Page Requests vs Selectivity (5000)

Mean CPU Time			
SELECTIVITY (%)	1000		
	INDEX	AFTER SORT	BEFORE SORT
1	0.032	0.531	0.150
10	0.106	0.588	0.337
20	0.188	0.652	0.464
30	0.270	0.716	0.591
40	0.352	0.780	0.718
50	0.434	0.844	0.940
60	0.517	0.908	1.070
70	0.599	0.972	1.200
80	0.681	1.036	1.330
90	0.763	1.100	1.460
100	0.845	1.164	1.589

Table A1.11 CPU vs Selectivity (1000)

Mean CPU Time			
SELECTIVITY (%)	5000		
	INDEX	AFTER SORT	BEFORE SORT
1	0.065	1.812	0.282
10	0.434	2.100	1.031
20	0.845	2.419	1.680
30	1.255	2.739	2.329
40	1.666	3.059	2.978
50	2.076	3.379	3.627
60	2.487	3.699	4.442
70	2.897	4.019	5.105
80	3.308	4.338	5.768
90	3.718	4.658	6.431
100	4.129	4.978	7.094

Table A1.12 CPU vs Selectivity (5000)

1.3. Type 3 Queries

Mean Response Time				
SELECTIVITY (%)	1000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	0.319	0.535	0.713	1.789
10	0.736	0.743	0.763	1.844
20	0.953	0.965	0.818	1.906
30	1.178	1.182	0.874	1.968
40	1.393	1.398	0.930	2.039
50	1.704	1.703	0.986	2.091
60	1.931	1.910	1.041	2.153
70	2.149	2.113	1.097	2.215
80	2.376	2.317	1.153	2.276
90	3.080	2.979	1.208	2.338
100	3.410	3.480	1.264	2.400

Table A1.13 Response Time vs Selectivity (1000)

Mean Response Time				
SELECTIVITY (%)	5000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	0.937	2.037	54.189	16.758
10	7.017	3.187	54.440	17.036
20	14.201	4.886	54.718	17.345
30	22.046	7.376	54.997	17.653
40	29.587	9.456	55.275	17.962
50	37.117	11.466	55.554	18.270
60	46.750	15.534	55.832	18.579
70	54.489	17.737	56.111	18.887
80	62.320	19.900	56.389	19.196
90	70.192	22.135	56.668	19.504
100	78.007	25.270	56.946	19.813

Table A1.14 Response Time vs Selectivity (5000)

Confidence Intervals (95%)				
SELECTIVITY (%)	1000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	±0.009	±0	±0	±0.004
10	±0.004	±0.001	±0	±0.004
20	±0.004	±0.001	±0	±0.004
30	±0.004	±0.002	±0	±0.004
40	±0.004	±0.002	±0	±0.004
50	±0.004	±0.007	±0	±0.004
60	±0.004	±0.004	±0	±0.004
70	±0.004	±0.005	±0	±0.004
80	±0.004	±0.004	±0	±0.004
90	±0.004	±0.006	±0	±0.004
100	±0.005	±0.004	±0	±0.004

Table A1.15 Confidence Interval for Response Time vs Selectivity (1000)

Confidence Intervals (95%)				
SELECTIVITY (%)	5000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	±0.020	±0.004	±0.251	±0.004
10	±0.092	±0.001	±0.251	±0.004
20	±0.111	±0.007	±0.251	±0.004
30	±0.136	±0.010	±0.251	±0.004
40	±0.158	±0.013	±0.251	±0.004
50	±0.157	±0.016	±0.251	±0.004
60	±0.152	±0.012	±0.251	±0.004
70	±0.189	±0.016	±0.251	±0.004
80	±0.241	±0.012	±0.251	±0.004
90	±0.260	±0.013	±0.251	±0.004
100	±0.259	±0.004	±0.251	±0.004

Table A1.16 Confidence Interval for Response Time vs Selectivity (5000)

Mean Page I/O				
SELECTIVITY (%)	1000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	12	24	24	84
10	28	28	24	84
20	36	36	24	84
30	44	44	24	84
40	52	52	24	84
50	70	70	24	84
60	80	80	24	84
70	90	90	24	84
80	100	100	24	84
90	113	112	24	84
100	126	124	24	84

Table A1.17 Page Requests vs Selectivity (1000)

Mean Page I/O				
SELECTIVITY (%)	5000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	46	104	3103	900
10	375	150	3103	900
20	744	205	3103	900
30	1164	318	3103	900
40	1566	409	3103	900
50	1964	495	3103	900
60	2481	696	3103	900
70	2894	796	3103	900
80	3312	898	3103	900
90	3731	1000	3103	900
100	4151	1100	3103	900

Table A1.18 Page Requests vs Selectivity (5000)

Mean CPU Time				
SELECTIVITY (%)	1000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	0.134	0.150	0.305	0.553
10	0.339	0.337	0.355	0.608
20	0.487	0.464	0.410	0.670
30	0.634	0.591	0.466	0.732
40	0.782	0.718	0.522	0.794
50	1.024	0.940	0.578	0.855
60	1.174	1.070	0.633	0.917
70	1.324	1.200	0.689	0.979
80	1.475	1.330	0.745	1.040
90	1.625	1.460	0.800	1.102
100	1.775	1.589	1.264	1.164

Table A1.19 CPU vs Selectivity (1000)

Mean CPU Time				
SELECTIVITY (%)	5000			
	INDEX A_i	BEFORE SORT	INDEX A_j	AFTER SORT
1	0.183	0.282	1.442	1.924
10	1.024	1.031	1.693	2.201
20	1.775	1.680	1.971	2.510
30	2.526	2.329	2.250	2.818
40	3.277	2.978	2.528	3.127
50	4.028	3.627	2.807	3.435
60	4.945	4.442	3.085	3.744
70	5.710	5.105	3.364	4.052
80	6.475	5.768	3.642	4.361
90	7.240	6.431	3.921	4.669
100	8.005	7.094	4.199	4.978

Table A1.20 CPU vs Selectivity (5000)

Appendix A2

Join Results

The performance results obtained for the joins are listed in this appendix.

2.1. Two Large Relations

2.1.1. 1-to-1

Mean Response Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	7.449	6.303	9.792	8.665	8.391
0.0001	7.522	6.375	10.385	9.054	8.670
0.0002	7.601	6.455	10.477	9.383	8.754
0.0003	7.688	6.542	10.559	9.642	8.836
0.0004	7.768	6.622	10.641	9.944	8.919
0.0005	7.846	6.700	10.723	10.272	9.001
0.0006	7.928	6.781	10.805	10.584	9.083
0.0007	8.005	6.859	10.887	10.850	9.165
0.0008	8.087	6.940	10.969	11.124	9.247
0.0009	8.166	7.020	11.051	11.343	9.329
0.001	8.250	7.104	11.134	11.527	9.411

Table A2.1 Response Time vs Join Selectivity

Confidence Intervals (95%)					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	±0.007	±0.005	±0.010	±0.005	±0.005
0.0001	±0.007	±0.004	±0.009	±0.021	±0.006
0.0002	±0.007	±0.004	±0.005	±0.031	±0
0.0003	±0.007	±0.004	±0.005	±0.035	±0
0.0004	±0.007	±0.004	±0.005	±0.050	±0
0.0005	±0.007	±0.005	±0.005	±0.043	±0
0.0006	±0.007	±0.004	±0.005	±0.044	±0
0.0007	±0.007	±0.004	±0.005	±0.055	±0
0.0008	±0.007	±0.004	±0.005	±0.068	±0
0.0009	±0.007	±0.004	±0.005	±0.073	±0
0.001	±0.007	±0.004	±0.005	±0.063	±0

Table A2.2 Confidence Interval for Response Time vs Join Selectivity

Mean Page I/O					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	189	128	96	36	44
0.0001	189	127	127	54	44
0.0002	189	127	127	69	44
0.0003	189	128	127	79	44
0.0004	189	128	127	92	44
0.0005	189	128	127	107	44
0.0006	189	128	127	120	44
0.0007	189	127	127	131	44
0.0008	189	127	127	142	44
0.0009	189	127	127	150	44
0.001	189	128	127	156	44

Table A2.3 Page Requests vs Join Selectivity

Mean CPU Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	4.632	4.333	8.351	8.056	7.850
0.0001	4.705	4.406	8.425	8.130	7.924
0.0002	4.786	4.487	8.507	8.212	8.006
0.0003	4.867	4.568	8.589	8.294	8.088
0.0004	4.947	4.648	8.671	8.376	8.171
0.0005	5.028	4.729	8.754	8.458	8.253
0.0006	5.109	4.810	8.836	8.540	8.335
0.0007	5.189	4.890	8.918	8.623	8.417
0.0008	5.270	4.971	9.000	8.705	8.499
0.0009	5.351	5.052	9.082	8.787	8.581
0.001	5.431	5.132	9.164	8.869	8.663

Table A2.4 CPU vs Join Selectivity

2.1.2. 1 to n

Mean Response Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	7.450	6.304	9.815	8.665	8.391
0.0001	7.524	6.377	10.360	9.079	8.669
0.0002	7.606	6.460	10.466	9.411	8.754
0.0003	7.692	6.546	10.551	9.685	8.836
0.0004	7.774	6.627	10.636	9.987	8.919
0.0005	7.854	6.708	10.718	10.282	9.001
0.0006	7.937	6.791	10.801	10.598	9.083
0.0007	8.023	6.876	10.884	10.833	9.165
0.0008	8.106	6.959	10.967	11.088	9.247
0.0009	8.189	7.043	11.049	11.264	9.329
0.001	8.294	7.148	11.132	11.477	9.411

Table A2.5 Response Time vs Join Selectivity

Confidence Intervals (95%)					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	±0.007	±0.004	±0.019	±0.005	±0.005
0.0001	±0.007	±0.004	±0.019	±0.028	±0.006
0.0002	±0.007	±0.004	±0.009	±0.029	±0
0.0003	±0.006	±0.004	±0.008	±0.043	±0
0.0004	±0.006	±0.004	±0.007	±0.047	±0
0.0005	±0.007	±0.004	±0.007	±0.065	±0
0.0006	±0.006	±0.005	±0.007	±0.073	±0
0.0007	±0.007	±0.004	±0.007	±0.083	±0
0.0008	±0.007	±0.004	±0.006	±0.064	±0
0.0009	±0.006	±0.004	±0.006	±0.080	±0
0.001	±0.006	±0.005	±0.005	±0.070	±0

Table A2.6 Confidence Interval for Response Time vs Join Selectivity

Mean Page I/O					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	189	128	98	36	32
0.0001	189	127	125	56	44
0.0002	189	127	127	71	44
0.0003	189	128	127	82	44
0.0004	189	128	127	95	44
0.0005	189	127	127	107	44
0.0006	189	127	127	121	44
0.0007	189	128	127	130	44
0.0008	189	128	127	140	44
0.0009	189	128	127	146	44
0.001	189	128	127	153	44

Table A2.7 Page Requests vs Join Selectivity

Mean CPU Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	4.633	4.334	8.351	8.056	7.850
0.0001	4.707	4.408	8.425	8.130	7.924
0.0002	4.790	4.491	8.507	8.212	8.006
0.0003	4.873	4.574	8.589	8.294	8.088
0.0004	4.955	4.656	8.671	8.376	8.171
0.0005	5.038	4.739	8.754	8.458	8.253
0.0006	5.120	4.822	8.836	8.540	8.335
0.0007	5.204	4.905	8.918	8.623	8.417
0.0008	5.286	4.987	9.000	8.705	8.499
0.0009	5.368	5.069	9.082	8.787	8.581
0.001	5.471	5.172	9.164	8.869	8.663

Table A2.8 CPU vs Join Selectivity

2.1.3. n to m

Mean Response Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.0001	7.523	6.378	9.887	8.739	8.465
0.001	8.271	7.123	11.115	9.818	9.406
0.002	9.097	7.948	11.951	10.896	10.232
0.003	9.927	8.776	12.778	11.943	11.053
0.004	10.757	9.605	13.601	12.958	11.874
0.005	11.582	10.436	14.418	14.035	12.695
0.006	12.409	11.263	15.238	15.127	13.516
0.007	13.234	12.086	16.061	16.116	14.337
0.008	14.061	12.913	16.882	17.095	15.158
0.009	14.892	13.743	17.703	18.026	15.979
0.01	15.717	14.568	18.525	18.929	16.800

Table A2.9 Response Time vs Join Selectivity

Confidence Intervals (95%)					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.0001	±0.007	±0.004	±0.019	±0.005	±0.005
0.001	±0.008	±0.005	±0.011	±0.028	±0.005
0.002	±0.007	±0.005	±0.008	±0.038	±0
0.003	±0.007	±0.005	±0.006	±0.054	±0
0.004	±0.007	±0.004	±0.005	±0.062	±0
0.005	±0.008	±0.006	±0.004	±0.079	±0
0.006	±0.007	±0.005	±0.004	±0.052	±0
0.007	±0.006	±0.005	±0.004	±0.065	±0
0.008	±0.004	±0.004	±0.004	±0.075	±0
0.009	±0.005	±0.004	±0.005	±0.086	±0
0.01	±0.006	±0.005	±0.004	±0.080	±0

Table A2.10 Confidence Interval for Response Time vs Join Selectivity

Mean Page I/O					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.0001	189	128	98	36	32
0.001	189	127	126	56	44
0.002	189	127	127	71	44
0.003	189	127	128	84	44
0.004	189	128	128	96	44
0.005	189	128	128	111	44
0.006	189	128	127	127	44
0.007	189	128	128	137	44
0.008	189	127	128	146	44
0.009	189	128	128	152	44
0.01	189	127	128	157	44

Table A2.11 Page Requests vs Join Selectivity

Mean CPU Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.0001	4.707	4.408	8.425	8.130	7.924
0.001	5.451	5.152	11.115	8.869	8.663
0.002	6.278	5.980	11.951	9.690	9.484
0.003	7.106	6.807	12.778	10.511	10.305
0.004	7.933	7.634	13.601	11.332	11.126
0.005	8.761	8.462	14.418	12.153	11.947
0.006	9.589	9.290	15.238	12.974	12.768
0.007	10.415	10.116	16.061	13.795	13.589
0.008	11.243	10.944	16.882	14.616	14.410
0.009	12.069	11.770	17.703	15.437	15.231
0.01	12.897	12.598	18.525	16.258	16.052

Table A2.12 CPU vs Join Selectivity

2.2. One Large and One Small Relation

2.2.1. 1 to 1

Mean Response Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	4.827	4.028	5.035	4.449	4.310
0.0001	4.860	4.061	5.250	4.665	4.526
0.0002	4.899	4.100	5.315	4.728	4.589
0.0003	4.939	4.140	5.361	4.774	4.635
0.0004	4.980	4.180	5.402	4.815	4.676
0.0005	5.021	4.221	5.444	4.856	4.717
0.0006	5.062	4.260	5.487	4.899	4.760
0.0007	5.101	4.301	5.528	4.940	4.801
0.0008	5.142	4.342	5.571	4.983	4.844
0.0009	5.180	4.381	5.613	5.024	4.885
0.001	5.218	4.419	5.654	5.065	4.926

Table A2.13 Response Time vs Join Selectivity

Confidence Intervals (95%)					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	±0.004	±0.004	±0.006	±0.005	±0.005
0.0001	±0.004	±0.004	±0.009	±0.008	±0.008
0.0002	±0.004	±0.004	±0.004	±0.004	±0.004
0.0003	±0.004	±0.004	±0.004	±0.003	±0.003
0.0004	±0.004	±0.004	±0.004	±0.003	±0.003
0.0005	±0.004	±0.004	±0.004	±0.003	±0.003
0.0006	±0.005	±0.005	±0.005	±0.004	±0.004
0.0007	±0.004	±0.004	±0.005	±0.004	±0.004
0.0008	±0.005	±0.006	±0.005	±0.004	±0.004
0.0009	±0.004	±0.005	±0.005	±0.004	±0.004
0.001	±0.004	±0.004	±0.004	±0.004	±0.004

Table A2.14 Confidence Interval for Response Time vs Join Selectivity

Mean Page I/O					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002 145	105	52	24	22	
0.0001	145	105	63	35	33
0.0002	145	105	64	36	34
0.0003	145	105	64	36	34
0.0004	145	105	64	36	34
0.0005	145	105	64	36	34
0.0006	145	105	64	36	34
0.0007	145	105	64	36	34
0.0008	145	105	64	36	34
0.0009	145	105	64	36	34
0.001	144	104	64	36	34

Table A2.15 Page Requests vs Join Selectivity

Mean CPU Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	2.716	2.446	4.311	4.044	3.939
0.0001	2.749	2.479	4.343	4.077	3.972
0.0002	2.789	2.519	4.384	4.118	4.013
0.0003	2.829	2.559	4.426	4.159	4.054
0.0004	2.870	2.599	4.467	4.200	4.095
0.0005	2.910	2.640	4.508	4.241	4.136
0.0006	2.950	2.680	4.549	4.282	4.177
0.0007	2.991	2.721	4.590	4.323	4.218
0.0008	3.031	2.761	4.631	4.364	4.259
0.0009	3.072	2.801	4.672	4.403	4.300
0.001	3.112	2.842	4.713	4.446	4.341

Table A2.16 CPU vs Join Selectivity

2.2.2. n to 1

Mean Response Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	4.828	4.030	4.919	4.333	4.194
0.0001	4.861	4.062	5.010	4.424	4.285
0.0002	4.900	4.102	5.110	4.523	4.384
0.0003	4.941	4.145	5.201	4.614	4.475
0.0004	4.982	4.186	5.276	4.689	4.550
0.0005	5.021	4.226	5.347	4.760	4.621
0.0006	5.062	4.266	5.407	4.820	4.681
0.0007	5.102	4.304	5.464	4.877	4.738
0.0008	5.141	4.343	5.519	4.931	4.792
0.0009	5.182	4.384	5.573	4.984	4.845
0.001	5.222	4.419	5.624	5.035	4.896

Table A2.17 Response Time vs Join Selectivity

Confidence Intervals (95%)					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	±0.004	±0.004	±0.002	±0	±0
0.0001	±0.004	±0.005	±0.005	±0.005	±0.005
0.0002	±0.004	±0.005	±0.006	±0.005	±0.005
0.0003	±0.004	±0.006	±0.010	±0.009	±0.009
0.0004	±0.004	±0.005	±0.012	±0.012	±0.012
0.0005	±0.004	±0.006	±0.011	±0.010	±0.010
0.0006	±0.004	±0.005	±0.009	±0.009	±0.009
0.0007	±0.004	±0.007	±0.012	±0.011	±0.011
0.0008	±0.004	±0.006	±0.010	±0.009	±0.009
0.0009	±0.004	±0.005	±0.008	±0.008	±0.008
0.001	±0.004	±0.008	±0.008	±0.008	±0.008

Table A2.18 Confidence Interval for Response Time vs Join Selectivity

Mean Page I/O					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002 145	105	45	17	15	
0.0001	145	105	48	20	18
0.0002	145	105	52	24	22
0.0003	145	105	55	27	25
0.0004	145	105	57	29	27
0.0005	145	105	59	31	29
0.0006	145	105	60	32	30
0.0007	145	105	61	33	31
0.0008	145	105	61	33	31
0.0009	145	105	62	34	32
0.001	145	104	63	35	33

Table A2.19 Page Requests vs Join Selectivity

Mean CPU Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	2.717	2.446	4.311	4.044	3.939
0.0001	2.749	2.478	4.343	4.077	3.972
0.0002	2.789	2.519	4.384	4.118	4.013
0.0003	2.830	2.560	4.426	4.159	4.054
0.0004	2.870	2.600	4.467	4.200	4.095
0.0005	2.911	2.640	4.508	4.241	4.136
0.0006	2.951	2.681	4.549	4.282	4.177
0.0007	2.990	2.720	4.590	4.323	4.218
0.0008	3.031	2.761	4.631	4.364	4.259
0.0009	3.071	2.801	4.672	4.405	4.300
0.001	3.111	2.841	4.713	4.446	4.341

Table A2.20 CPU vs Join Selectivity

2.2.3. n to m

Mean Response Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.0002	4.902	4.103	5.109	4.523	4.384
0.001	5.233	4.435	5.620	5.035	4.896
0.002	5.646	4.849	6.051	5.464	5.325
0.003	6.062	5.262	6.466	5.879	5.740
0.004	6.476	5.677	6.877	6.290	6.151
0.005	6.888	6.091	7.288	6.700	6.561
0.006	7.301	6.503	7.699	7.111	6.972
0.007	7.714	6.917	8.109	7.521	7.382
0.008	8.126	7.328	8.520	7.932	7.793
0.009	8.542	7.743	8.931	8.342	8.203
0.01	8.953	8.152	9.342	8.753	8.614

Table A2.21 Response Time vs Join Selectivity

Confidence Intervals (95%)					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00001	±0.004	±0.004	±0.006	±0.005	±0.005
0.0001	±0.004	±0.005	±0.008	±0.008	±0.008
0.0002	±0.004	±0.005	±0.005	±0.005	±0.005
0.0003	±0.005	±0.007	±0.002	±0	±0
0.0004	±0.005	±0.006	±0.002	±0	±0
0.0005	±0.004	±0.005	±0.002	±0	±0
0.0006	±0.004	±0.005	±0.002	±0	±0
0.0007	±0.005	±0.004	±0.002	±0	±0
0.0008	±0.005	±0.006	±0.002	±0	±0
0.0009	±0.006	±0.006	±0.001	±0	±0
0.001	±0.005	±0.005	±0	±0	±0

Table A2.22 Confidence Interval for Response Time vs Join Selectivity

Mean Page I/O					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.00002	145	105	52	24	22
0.0001	145	105	63	35	33
0.0002	145	105	64	36	34
0.0003	145	105	64	36	34
0.0004	145	105	64	36	34
0.0005	145	105	64	36	34
0.0006	145	105	64	36	34
0.0007	145	105	64	36	34
0.0008	145	105	64	36	34
0.0009	145	105	64	36	34
0.001	144	104	64	36	34

Table A2.23 Page Requests vs Join Selectivity

Mean CPU Time					
JOIN SELECTIVITY FACTOR	MN	MO	MI	NB	NI
0.0002	2.791	2.521	4.385	4.118	4.013
0.001	3.122	2.851	4.713	4.446	4.341
0.002	3.535	3.265	5.123	4.857	4.752
0.003	3.950	3.680	5.534	5.267	5.162
0.004	4.363	4.093	5.944	5.678	5.573
0.005	4.776	4.506	6.355	6.088	5.983
0.006	5.190	4.920	6.765	6.499	6.394
0.007	5.604	5.334	7.176	6.909	6.804
0.008	6.016	5.746	7.586	7.320	7.215
0.009	6.431	6.161	7.997	7.730	7.625
0.01	6.845	6.575	8.407	8.141	8.036

Table A2.24 CPU vs Join Selectivity